Meta Reinforcement Learning through Memory

Emilio Parisotto

September 2021
CMU-ML-21-112

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported by the Office of Naval Research awards N000141613074 and N000141812861, and a grant from Apple.

Keywords: deep reinforcement learning, deep learning, meta-learning

Meta Reinforcement Learning through Memory

Thesis

Emilio Parisotto

Abstract

Modern deep reinforcement learning (RL) algorithms, despite being at the forefront of artificial intelligence capabilities, typically require a prohibitive amount of training samples to reach a human-equivalent level of performance. This severe data inefficiency is the major obstruction to deep RL's practical application: it is often near impossible to apply deep RL to any domain without at least a simulator available. Motivated to address this critical data inefficiency, in this thesis we work towards the design of meta-learning agents that are capable of rapidly adapting to new environments. In contrast to standard reinforcement learning, meta-learning learns over distributions of environments, from which specific tasks are sampled and with which the meta-learner is directly optimized to improve the speed of policy improvement on. By exploiting a distribution of tasks which share common substructure with the tasks of interest, the meta-learner can adjust its own inductive biases to enable rapid adaptation at test time.

This thesis focuses on the design of meta-learning algorithms which exploit memory as the main mechanism driving rapid adaptation in novel environments. Meta-learning with inter-episodic memories are a class of meta-learning methods that leverage a memory architecture conditioned on the entire interaction history of a particular environment to produce a policy. The learning dynamics driving policy improvement in a particular task are thus subsumed by the computational process of the sequence model, essentially offloading the design of the learning algorithm to the architecture. While conceptually straightforward, meta-learning using inter-episodic memory is highly effective and remains a state-of-the-art method.

We present and discuss several techniques for meta-learning through memory. The first part of the thesis focuses on the "embodied" class of environments, where an agent has a physical manifestation in an environment resembling the natural world. We exploit this highly structured set of environments to work towards the design of a monolithic embodied agent architecture that has the capabilities of rapid memorization, planning and state inference. In the second part of the thesis, we move to focus on methods that apply in general environments without strong common substructure. First, we re-examine the modes of interaction a meta-learning agent has with the environment: proposing to replace the typically sequential processing of interaction history with a concurrent execution framework where multiple agents act in the environment in parallel. Next, we discuss the use of a general and powerful sequence model for inter-episodic memory, the gated transformer, demonstrating large improvements in performance and data efficiency. Finally, we develop a method that significantly reduces the training cost and acting latency of transformer models in (meta-)reinforcement learning settings, with the aim to both (1) make their use more widespread within the research community, and, (2) unlock their use in real-time and latency-constrained applications, such as in robotics.

Contents

1 Introduction **5**
1.1 Notation . . . 6
1.1.1 Meta-Learning . . . 7
1.1.2 Embodied Environments . . . 8
1.2 Overview of Completed Work . . . 8
1.2.1 Part 1 - Meta Reinforcement Learning in Embodied Domains . . . 9
1.2.2 Part 2 - Meta Reinforcement Learning through Memory . . . 11
1.3 Summary . . . 13

I Meta Reinforcement Learning in Embodied Environments **15**

2 Neural Map: Structured Inter-Episodic Memory **16**
2.1 Background & Related Work . . . 16
2.2 Neural Map . . . 17
2.3 Ego-centric Neural Map . . . 19
2.4 Controller (Ego-)Neural Map . . . 20
2.5 Experiments . . . 20
2.5.1 2D Goal-Search Environment . . . 20
2.5.2 3D Doom Environment Description . . . 22
2.5.3 3D Doom Model Description . . . 23
2.5.4 Robustness to Odometric Noise . . . 24
2.5.5 Samples of Context Read Distribution . . . 24
2.5.6 Backtracking . . . 25
2.6 Discussion . . . 25

3 Planning and State Inference in Embodied Inter-Episodic Memories **28**
3.1 Background & Related Work . . . 29
3.2 Environments and Maze Transition Types . . . 30
3.2.1 Domain Descriptions . . . 31
3.2.2 Transition Types . . . 32
3.3 Gated Path Planning Networks . . . 32
3.3.1 Planning Experiments . . . 33
3.4 Active Neural Localizer . . . 38
3.4.1 Model Architecture . . . 40
3.4.2 Model Components . . . 40
3.4.3 Baselines . . . 40
3.4.4 Results . . . 41
3.5 Discussion . . . 43

4 Towards End-to-End Simultaneous Localization and Mapping **46**
4.1 Background & Related Work . . . 47
4.2 Neural Graph Optimizer . . . 49
4.2.1 Local Pose Estimation Network . . . 49
4.2.2 Pose Aggregation . . . 49
4.2.3 Neural Graph Optimization . . . 50
4.3 Experiments . . . 53

4.3.1 2D Environment 53
4.3.2 3D Environment 54
4.4 Discussion 55

II Meta Reinforcement Learning through Memory 59

5 Concurrent Episodic Meta Reinforcement Learning 60
5.1 Background & Related Work 61
5.2 Concurrent Episodic Meta Reinforcement Learning 61
5.2.1 Communication Architecture 63
5.2.2 Reward Sharing Schemes 64
5.2.3 Divergence Losses 65
5.3 Experiments 65
5.4 Discussion 69

6 Transformers for Meta RL 70
6.1 Background & Related Work 71
6.1.1 Transformer Architecture and Variants 72
6.2 Gated Transformer Architectures 73
6.2.1 Identity Map Reordering 73
6.2.2 Gating Layers 74
6.2.3 Motivating the Gated Transformer-XL Architecture 75
6.3 Methodology 76
6.3.1 Environment Details 76
6.3.2 Agent Details 77
6.4 Experiments 81
6.4.1 Transformer as Effective RL Memory Architecture 81
6.4.2 Scaling with Memory Horizon 82
6.4.3 Gating Variants + Identity Map Reordering 83
6.4.4 Atari-57 Results 85
6.4.5 Motivating the Identity Map Re-ordering 86
6.5 Discussion 87

7 Efficient Transformers for Meta RL 89
7.1 Background & Related Work 90
7.2 Actor-Learner Distillation 91
7.3 Distributed Actor-Learner Distillation 93
7.3.1 Improving Distill / RL Step Ratio 93
7.4 Methodology 94
7.4.1 Algorithm Details 94
7.4.2 Baselines 95
7.4.3 Compute Details 95
7.4.4 Common Experimental Details 95
7.5 Experiments 96
7.5.1 I-Maze 96
7.5.2 Meta-Fetch 97
7.6 Discussion 99

8 Conclusion 100
8.1 Future Directions 101
8.1.1 Scaling Meta-Learning through Memory: Model Capacity 101
8.1.2 Scaling Meta-Learning through Memory: Context Length 101
8.1.3 Improved Efficiency for Large-Scale Transformers 102
8.1.4 General Sequence Models for Embodied Environments 102

Chapter 1

Introduction

Humans and other animals can rapidly integrate information to perform tasks within only a handful of interactions. The ability of natural learning systems to generalize to novel situations is, however, in astounding contrast to what current artificial systems are capable of. Despite being at the forefront of artificial intelligence capabilities, modern deep reinforcement learning algorithms can take in excess of hundreds of thousands of trials to learn even a single task [137] [135]. Naively leveraging a multitasking objective does not remedy this situation, and only introduces further complexities into the proper functioning of the algorithm such as difficult optimization and the tendency for the agent to forget and/or overemphasize certain tasks [148]. The inability of reinforcement learning agents to rapidly generalize to new situations therefore presents itself as a critical obstruction to the application of AI to real-world domains.

Rapid generalization within novel environments can be facilitated by equipping the agent with the proper inductive biases. These biases can be the fixed architectural components of the agent's model, i.e. the use of convolutions for image data or LSTMs for sequence data, or they can be obtained themselves through data-driven processes. The data-driven approach is especially promising as it can enable an agent to tailor it's own inductive biases to different environments in an online fashion, as it is interacting with them. One of the earliest methodologies [205][168][181] that has recently gained increasing empirical success [14][210][49][161][56] is to have the agent's own learning process be data-driven, i.e. having the agent learn-to-learn, or meta-learn [180]. In contrast to the generality of reinforcement learning algorithms defined over broad classes of environments (i.e. all Markov Decision Processes), meta-learning allows the agent to adjust its learning process to specialize to a more narrow distribution of environments with an underlying exploitable structure; sacrificing complete generality for rapid adaptation to a set of target environments.

Meta-learning processes can generally be described under the framework of an agent having separate outer-loop (*slow*) and inner-loop (*fast*) learning systems [16]. The *fast* learning system represents the agent's learning process that occurs during interaction with a particular instance of an environment, determining the agent's online behavior. This learning system is the one which is adapted specifically to the types of environments the agent designer is targeting for rapid adaptation. At the opposite end of the spectrum of temporal horizons, the *slow* learning system represents the complementary part of the agent taking experiences aggregated across many different trials in order to tailor the fast learning system's inductive biases to the distribution of target environments. This slow system is typically driven by standard reinforcement learning [49][210][161][56] or self-supervised objectives [14], albeit at much longer horizons than usual as meta-episodes last a multiple of the individual horizons of environment instances.

One straightforward but particularly effective meta-learning approach is to endow the agent with an inter-episodic memory [49][210][14][161], or a memory that lasts across an agent's learning interactions within a particular environment. Here, the fast learning system is a sequential model, the exact form and architecture of which can vary widely, and the slow learning system is an algorithm optimizing an RL or self-supervised objective that adapts the parameters of the sequential model across many different environment instances. This inter-episodic memory framework transforms the meta-learning problem into one of finding effective and powerful sequential model architectures. While straightforward to conceptualize, the choice of memory architecture is not a trivial decision and, as will be demonstrated in this thesis, can have a profound impact on the learning performance of the meta-learning agent.

There exist a wide distribution of techniques involving meta-learning using an inter-episodic memory, with each varying in the form of memory and in ways the memory influences behavior during learning. *Episodic control*, a form of non-parametric inter-episodic memory first described in the cognitive sciences literature [50][120], is a memory architecture that largely exploits Markov Decision Process structure to enable fast and data-efficient learning [120][79][14][161][169]. Methods for *episodic control* essentially utilize an associative memory where, given a new observation, the closest observations in past experience are recalled along with the reward statistics of that past experience, i.e. the return obtained last time the agent was in a similar state. This non-parametric memory enables very fast instance-based learning where, when interacting with relatively deterministic dynamics, a handful of similar past experiences can be adequate to acquire a reasonable assessment of obtainable future reward. Recently this concept has been scaled to the high-dimensional environments typical of deep reinforcement learning applications, with the episodic memory being implemented either as a k-Nearest-Neighbor model (kNN) using fixed random projections of observations as features [14], or a soft-kNN where kNN features are trained alongside the policy using RL objectives [161]. In this setting, the kNN memory look-up functions as re-acquiring stored MDP statistics, i.e. estimates of return statistics for different actions, and the majority of learning the system undertakes is mainly to improve observation featurization so as to enable accurate, generalizable recall. In psychology and cognitive sciences, episodic memories have emerged as an accurate model of certain animal learning behaviours, suggesting a potential implementation of a similar mechanism in natural systems [169][16].

An alternative parametric approach to inter-episodic memories involves recurrent neural networks whose recurrent state does not reset at episode boundaries [49][210]. These parametric approaches are in a sense more general because they (1) do not restrain the recalled information to be reward-based statistics, and (2) can in theory learn any algorithm, including the standard RL ones episodic memory leverages (assuming the particular recurrent network model is Turing-complete [184]). The previously described approaches are not necessarily mutually exclusive: Parametric and non-parametric inter-episodic memories have also recently been combined, where non-parametric recall is used to reinstate the activations of a recurrent policy, based on the closeness of the current observation history to past experiences [170].

A notable counterpart to inter-episodic memory utilizes gradient descent itself as an inductive bias for the meta-learning update rule of the system [128][5][56][144][175][59]. These optimization-based meta-learners differentiate through the optimization process, typically stochastic gradient descent or a historically-averaged variant, to obtain a descent direction on the inter-trial learning dynamics. Variants of this approach use the meta-gradient in different ways: optimizing the initial weight matrix to achieve optimal performance in as few steps as possible in a variety of environments [56], doing the same but using a more computationally efficient approximation to the second-order derivatives [144][59], optimizing a lower dimensional embedding vector that structurally parameterizes the full initial weight matrix of the network [175], optimizing all parameters of the learning process [5]. While these methods have emerged as a distinct branch of meta-learning, they are not directly competing with episodic memory approaches, and there is evidence that both approaches can be combined [59]. In this thesis, we do not focus on this approach and we believe that both methods can be developed independently as they each have separate strengths and disadvantages.

1.1 Notation

A Markov Decision Process (MDP) [163, 196] is a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \gamma, \mathcal{R}, \rho, \mathcal{H})$ where:

- $\mathcal{S}$ is a finite set of states.
- $\mathcal{A}$ is a finite set of actions.
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \Pi(\mathcal{S})$ defines the transition dynamics, e.g. $\mathcal{T}(s'|s, a)$ for states s, s', action a defines the probability of ending up in state s' when executing a in state s.
- $\gamma \in [0, 1]$ is a discount factor.
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a reward function which an agent tries to maximize.
- $\rho : \mathcal{S} \to \Pi(\mathcal{S})$ is the initial state distribution, defining where in the state-space the agent begins an episode.

- $\mathcal{H} \in \mathbb{N}$ is the time horizon of the MDP, after which an episode ends. For non-episodic environments, $\mathcal{H} = \infty$.

where $\Pi(\cdot)$ defines a probability distribution over its input set. MDPs are Markovian as knowing the current state is sufficient to know the distribution of next states, i.e. the path that was taken to the current state is irrelevant to decision making.

A stationary policy is a mapping from states to probabilities over actions $\pi : \mathcal{S} \to \Pi(\mathcal{A})$. We define the value of a policy $V^\pi(s_t)$ as the accumulated discounted expected future reward when starting at a state s_t and executing π onward:

$$V^\pi(s_t) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k s_{t+k} \middle| \begin{matrix} a_{t+k} \sim \pi(\cdot|s_{t+k}), \\ s_{t+k+1} \sim \mathcal{T}(\cdot|s_{t+k}, a_{t+k}) \end{matrix}\right] \tag{1.1}$$

We can additionally define a similar "action-value" function:

$$Q^\pi(s_t, a_t) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k s_{t+k} \middle| \begin{matrix} a_{t+k+1} \sim \pi(\cdot|s_{t+k+1}), \\ s_{t+k+1} \sim \mathcal{T}(\cdot|s_{t+k}, a_{t+k}) \end{matrix}\right] \tag{1.2}$$

The goal of reinforcement learning algorithms is to find the policy π^* that maximizes value over all states s: $\forall s$, $\pi^* = \arg\max_\pi V^\pi(s)$. It is guaranteed that π^* always exists [196]. The value and action-value functions can be decomposed into recursive forms called the Bellman Equations:

$$V^\pi(s_t) = \mathbb{E}\left[r(s_t, a_t) + \gamma V^\pi(s_{t+1}) \middle| \begin{matrix} a_t \sim \pi(\cdot|s_t), \\ s_{t+1} \sim \mathcal{T}(\cdot|s_t, a_t) \end{matrix}\right] \tag{1.3}$$

$$Q^\pi(s_t, a_t) = \mathbb{E}\left[r(s_t, a_t) + \gamma Q^\pi(s_{t+1}, a_{t+1}) \middle| \begin{matrix} s_{t+1} \sim \mathcal{T}(\cdot|s_t, a_t), \\ a_{t+1} \sim \pi(\cdot|s_{t+1}) \end{matrix}\right] \tag{1.4}$$

The optimal policy satisfies another recursive form called the Bellman Optimality Equation:

$$V^*(s_t) = \mathbb{E}\left[r(s_t, a_t) + \gamma \max_{a'} Q^*(s_{t+1}, a') \middle| \begin{matrix} a_t \sim \pi(\cdot|s_t), \\ s_{t+1} \sim \mathcal{T}(\cdot|s_t, a_t) \end{matrix}\right] \tag{1.5}$$

$$Q^*(s_t, a_t) = \mathbb{E}\left[r(s_t, a_t) + \gamma \max_{a'} Q^*(s_{t+1}, a') \middle| s_{t+1} \sim \mathcal{T}(\cdot|s_t, a_t)\right] \tag{1.6}$$

where $V^* = V^{\pi^*}$, $Q^* = Q^{\pi^*}$.

A Partially-Observable Markov Decision Process (POMDP) [97] is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \gamma, \mathcal{R}, \rho, \mathcal{H}, \Omega, \mathcal{O})$ where:

- $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \gamma, \mathcal{R}, \rho, \mathcal{H})$ is an MDP.
- Ω is a finite set of observations.
- $\mathcal{O} : \mathcal{S} \times \mathcal{A} \to \Pi(\Omega)$ defines an observation function where $\mathcal{O}(s', a, o)$ is the probability of observing o after executing action a and ending up in state s'.

The key distinction of POMDP from MDP is that, in POMDPs, the agent is unable to observe the current state, typically only having access to a sampled observation. Therefore in order to act optimally, an agent must try to infer its state using the full history of observations and actions taken, or a sufficient statistic of this history.

1.1.1 Meta-Learning

The class of meta-learning environments is primarily concerned with learning in resource constrained applications. This covers a wide variety of natural applications where environment interaction is expensive, due to the high cost of sensors or actuators (a robot costing several hundred thousand dollars), or where time is constrained (i.e. agent execution is occurring in real-time and therefore sampling millions of states is prohibitive). This can reduce the amount of data available for learning in a particular environment by orders of magnitude compared to the number of samples typically required to train deep reinforcement learning agents.

More formally, Meta Reinforcement Learning is concerned with the case where we are given a distribution $p : \mathcal{M} \to [0, 1]$ over tasks $\mathcal{M}^{\mathcal{T}} = \{\mathcal{M}_1, \ldots, \mathcal{M}_N\}$ and we want to learn how to act in any environment sampled from this distribution as quickly as possible. By learning to focus on only the variations between the environments in $\mathcal{M}^{\mathcal{T}}$, meta learning algorithms can exploit structure in the class of tasks to, in the most extreme case, learn a new $\mathcal{M}_i \sim p$ in only 1-10 interaction episodes. Adopting similar conventions to [56, 190], we can frame meta reinforcement learning as a solution to the following objective function:

$$\min_{\theta} \sum_{\mathcal{M}_i} \mathbb{E}_{\pi_{\Delta(\theta)}} \left[\mathcal{L}_{\mathcal{M}_i} \right] \tag{1.7}$$

where $\Delta(\theta)$ represents a few-shot update method which collects a limited amount of experience from each $\mathcal{M}_i$ to update θ. In this thesis, we focus on techniques where the few-shot update method $\Delta(\theta)$ is a sequence model processing an inter-episodic observation, action and reward history.

We formalize the terms for meta-learning with inter-episodic memory below:

- An environment **domain** or **class** $\mathcal{M}^{\mathcal{T}}$ is the set of tasks we want to meta-learn on.
- An environment/task **instance** is a single (PO)MDP in the environment domain, $\mathcal{M}_i \in \mathcal{M}^T$.
- An **interaction**, **episode** or **sub-episode** all represent an $\mathcal{H}$-length sequence of actions taken by the meta-learning policy in a single environment instance.
- A **meta-episode** refers to the total sequence of interactions with an environment instance.

Additionally, in the sequel, as the focus is on temporally-extended decision-making, i.e. the RL setting, we will not make an explicit distinction between "Meta-Learning" (often used to refer specifically to supervised learning domains) and "Meta-Reinforcement-Learning", and these two terms will be used interchangably.

Meta-learning settings can be formalized as POMDPs with a special structure: an agent is exposed to a collection of instance MDPs (or, more generally, POMDPs) which each have an independent task id, or context, associated with them. The goal of meta-learning is for the agent to accurately discover this context through interaction with the environment instance it's currently acting in, while potentially being limited in

1. the amount of interaction it has with each environment instance,
2. the total number of environment instances it can interact with during training.

Given that meta-learning is a specific type of POMDP, techniques developed for solving general POMDPs, such as belief-space algorithms [122, 123, 222, 158] or recurrent networks [80], can therefore be applied to the meta-learning setting with few changes. Additionally, recent work [68] has made more explicit use of this connection between POMDP and meta-learning generalization, defining an "Epistemic POMDP" corresponding to the data-limited meta-learning setting where only a few environment instances are accessible during training, and leveraging Bayesian RL approaches, originally used for belief-space planning in more general POMDPs, to motivate novel meta-learning algorithms.

1.1.2 Embodied Environments

In this thesis, a particular class of POMDPs we are interested in will be "embodied environments". In embodied environments, agents have a physical manifestation in an environment with dynamics resembling the real world. This environment domain covers a large number of real-world robotics applications, where the standard setting is an agent controlling a robot's actuators given observations obtained from on-board sensors. Partial-observability is inherent within this class of environments due to occlusion and changing environment conditions, potentially due to other actors within the environment. As this class of environments is situated within the natural world (or an approximation of it), there is available a large amount of prior knowledge to make use of in the design of agent architectures. For examples of shared structure, the transition dynamics follow the laws of classical mechanics, and the state and observation spaces have underlying euclidean geometric structure.

1.2 Overview of Completed Work

We partition the chapters in this thesis into two parts: the first which focuses solely on methods for meta-learning with inter-episodic memory in embodied environments, and the second which focuses on general (PO)MDP environment classes. In this section, we describe the contributions of this thesis within these two parts.

1.2.1 Part 1 - Meta Reinforcement Learning in Embodied Domains

In the first part of this thesis, we focus on meta-learning within the class of embodied agent environments, where there exist several strong priors we can exploit in the structure of the observation and state spaces. As will be demonstrated, leveraging knowledge about the geometric and physical structure of the natural world can enable the design of meta-learning agents that achieve rapid and efficient learning in embodied environments. In this section, we present several methods covering various capabilities that work towards an end-to-end monolithic agent architecture for few-shot learning in embodied environments.

1. A memory architecture, the Neural Map [152], that accomplishes end-to-end learning of structured environment representations where features are organized spatially; features close together on this spatial metric represent observations obtained at positions close together in the environment. We demonstrate that this architecture functions as a highly effective inter-episodic memory for meta-learning in embodied environments and will serve as the basis of the proposed monolithic architecture.

2. A state inference architecture, Active Neural Localization [25], that chooses actions that disambiguate an agent's location (equivalently the agent's state in embodied environments) given a map. This architecture also enables the agent to be trained to take actions which reduce entropy on the current location. Results demonstrate that the proposed architecture significantly outperforms previous methods in terms of performance and required computational cost.

3. A planning architecture, the Gated Path Planning Network [119] (GPPN), that is able to do planning on feature maps and location estimates produced by deep learning architectures such as the proposed Neural Map and Active Neural Localization. Through an extensive set of experimental evaluations, we show the GPPN reaches state-of-the-art performance for differentiable planning in embodied environments.

4. An end-to-end architecture trained to do simultaneous localization and mapping, the Neural Graph Optimizer [154]. The Neural Graph Optimizer's ability to produce locations without any map knowledge *a priori* is the last remaining component necessary for a self-contained agent architecture, as location estimates are required for the Neural Map. We show that the architecture achieves significant performance gains over previous baselines on an environment domain with high levels of perceptual aliasing.

Combined, these methods contain the components necessary to build a monolithic agent architecture capable of actively and simultaneously memorizing, planning and performing state inference in embodied environments. We now describe each of these completed chapters in more detail.

Chapter 2 - Neural Map: Structured Inter-Episodic Memory [152]

Mapping is the process of building a representation of the environment that could later be used for localization (i.e. state disambiguation) and planning. For embodied environments, an obvious choice of representation is the geometric structure of the environment. Traditionally, a large amount of work has gone into the development of methods to construct geometric maps from image and depth sensors [140][139][111][52][214]. These methods typically use mathematical models of light transport to register pixel or feature observations in 3D space [140][139][52].

While the traditional fully geometric methods can achieve a high degree of reconstruction accuracy within real-world environments, transforming 3D geometric information into behaviours for accomplishing tasks can require a substantial amount of hand-engineering. Alternatively, we design an architecture that leverages the spatial structure of the environment while making the representation stored learnable end-to-end alongside the task behavior. The Neural Map accomplishes this through registering feature observations obtained at a time step t with the agent's current location in the environment. By storing the features within a spatial metric, the agent gains the ability to reason about the spatial relationships between observations directly and more efficiently, i.e. convolutional networks can be used to process features organized spatially. We demonstrate that this spatial-map organization of feature information largely improves agent performance compared to more unstructured latent representations such as LSTM, while still enabling learning directly from reward signal.

Chapter 3 - Planning & State Inference in Embodied Inter-Episodic Memories[119, 25]

In this chapter, we present results that can augment the Neural Map memory described in the previous chapter with two additional core competencies: planning and state inference. We propose two submodules with these capabilities, the Gated Path Planning Network and Active Neural Localization, able to perform planning and state inference, respectively. We evaluate each submodule in isolation in order to ablate their performance against previous baselines directly. All submodules are evaluated within the same environment domains to demonstrate that their potential combination is simply a matter of composing them within a single end-to-end agent architecture.

Planning [119]: The act of planning in embodied environments reduces itself to the more specific subset of problems known as path planning, that is: given a goal location, what is the fastest route to achieve that target location. Given an estimate on the agent's location (potentially provided by the Active Neural Localizer) and an approximate map of the environment (potentially provided by the Neural Map), several traditional algorithms can be chosen to do action selection such as value iteration, but these are insufficient because they (1) require domain-specific knowledge (e.g. knowledge of the MDP transition dynamics), and (2) because they are non-differentiable, meaning the map representation is not provided a signal to adjust itself to be amenable to the planning submodule's computations, and vice versa. Our goal of a monolithic agent architecture therefore motivates us to seek out differentiable planning architectures capable of learning how to integrate map and location information automatically, even when given distributed representations for the map and location estimate. This can enable a planning mechanism tailored to the task at hand as the reward signal influences its operation, and further removes the requirement that intermediate representations in our agent architecture be interpretable, reducing the domain knowledge necessary for agent design.

The Gated Path Planning Network (GPPN) was developed specifically to tackle this challenge of planning with differentiable representations. The GPPN is based off the Value Iteration Network [198], an architecture that represents grid-based path planning using a differentiable neural network by assuming a local (i.e. Moore-neighbourhood) transition model. The VIN does planning by utilizing a weight-tied convolutional network on a spatially-organized grid to iteratively produce single-dimension feature maps that approximate value functions. The proposed GPPN replaces the hand-designed update equations with powerful gating mechanisms, resulting in an architecture that is far more reliable in optimization and performance than the VIN. We demonstrate that the GPPN largely outperforms the VIN on planning tasks in grid-world and complex 3D vision domains.

State Inference [25]: Localization, equivalently state inference in embodied domains, is the process of acquiring the agent's coordinates in the world given a map of the environment. Classical localization algorithms are based on Kalman Filters [100][187] or geometry-based methods [145][140]. Recently work has been done to leverage deep learning to produce robust and effective feature representations for use in localization methods [34][211][33]. Despite the use of end-to-end architectures in modern localization algorithms, a notably absent characteristic from recent work is the combination of action and localization, or active localization. An agent having the capability to choose actions to disambiguate its own location estimate can result in a potentially far more accurate localization than an agent purely choosing actions myopically.

To tackle the issue of simultaneously acting and localizing we propose Active Neural Localization [25], which does state estimation using an algorithm similar in conception to traditional approximate POMDP solvers. The key difference is that the Active Neural Localization replaces several components (such as the environment map, transition and observation model and policy), with differentiable sub-modules that allows learning directly from environment data, and thus enables composition in a monolithic end-to-end architecture. Active Neural Localization can learn to localize in environments where the POMDP parameters are inaccessible, and reduces agent designer involvement as approximations to these parameters can be learnt directly from reward signal. Furthermore, we design objectives which increase the amount of information-seeking actions the agent produces.

Chapter 4 - Towards End-to-End Simultaneous Localization and Mapping [154]

The inter-episodic memory at the basis of our proposed embodied agent architecture for meta-learning, the Neural Map, crucially relies on having a location estimate. As a location estimate depends on having a map and vice versa, we require some method capable of both localizing and mapping at the same time, solutions to which are termed Simultaneous Localization and Mapping

(SLAM) algorithms. SLAM is made difficult because mapping and localizing are highly intertwined: without an estimate of the camera's location, a map cannot be built, while simultaneously, without a map, localization cannot be performed. Furthermore, models which rely solely on local information [34][211] (i.e. ego-motion between temporally-contiguous frames) are susceptible to drift (the accumulation of small errors over time) and perceptual aliasing (the inability to disambiguate highly redundant scenes, i.e. every floor having an identical hallway). The difficulty and importance of SLAM algorithms has made it a highly active research field spanning decades [23], although the majority of this work has focused on hand-engineered pipelines that are non-differentiable.

While we can use any existing available SLAM solver [52][140][139] as an input to the Neural Map, there are several reasons why it is preferable to design a novel one: (1) because many of the traditional SLAM algorithms rely on hand-specified feature detectors, they can be brittle when scene statistics change (lighting, changes due to weather, etc.), and (2) the vast majority of these SLAM algorithms are non-differentiable, an obstruction to our stated goal of an end-to-end agent architecture. Therefore in this chapter, we design a differentiable submodule based off the graph-optimization-based class of SLAM algorithms [140][23][139]. Graph optimization SLAM algorithms construct a pose-graph where vertices are video frames and edge values are functions connecting and determining the strength of correspondences between frames, i.e. frame pairs with a high amount of similar features get high edge values. The pose graph is passed through an iterative optimization algorithm [111] that produces pose estimates that maximize the edge function values. The Neural Graph Optimizer replaces the pose graph construction and iterative optimization with a self-attention-based architecture, which is applied recurrently to produce a final refined estimate. By leveraging far longer-horizon temporal information than previous architectures [34][211], the Neural Graph Optimizer was shown to learn how to estimate locations more accurately than comparable end-to-end odometry and sequence-based methods.

1.2.2 Part 2 - Meta Reinforcement Learning through Memory

In the second part of the thesis, we move to the general meta-learning setting where there are no further assumptions on the types of (PO)MDPs the agent will encounter. This will require leveraging more universal structure in the methods we devise, for example how sequences are processed or even the method in which the agent interacts with its environment.

Chapter 5 - Concurrent Episodic Meta-Reinforcement Learning [150]

Meta-learning methods using inter-episodic memories typically process the sequence of interaction history with a particular environment instance into a conditional policy that predicts the next action [210][49][56]. While the sequential processing of interaction episodes within a single environment instance optimally uses the totality of information from that environment in deciding the next action, it is not always advantageous or viable to obtain a long history of episodes sequentially. For example in robotics, interacting with a robot is time-constrained as environment processing cannot occur faster than real-time. Processing data in parallel is a common workaround to applications where data collection is expensive: having multiple robots can enable a linear reduction in the time required to obtain a fixed number of samples. While this can come at the cost of modeling capacity (as each parallelized meta-policy may not have a complete observation history of the other meta-policies), parallelizing can often be a more cost-effective and practical solution.

Following along this type of reasoning, we present CEMRL [150], a meta-learning framework that parallelizes episode execution by formulating meta-learning as a multi-agent communication problem. In this framework, each episode of interaction with an environment instance is executed by a separate agent in parallel. Rapid meta-learning is achieved not purely through conditioning on observation history as typically done, but through the agents being interconnected through a differentiable communication architecture that enables the sharing of features from distinct episodes. In addition to being a more cost-effective solution for time-critical applications, parallelization additionally affords several advantages: it enables the design of reward sharing schemes that are well-suited to high-risk environments and auxiliary losses that encourage a diversity of behavior amongst the parallel policies. In a variety of challenging partially-observable environments, we evaluate several architectures in our paralellized meta-learning framework and demonstrate a consistent improvement over the more commonly used episodic sequence processing methods.

Chapter 6 - Transformers for Meta RL [155]

As described in previous sections, the specialized design of architectures suited to certain environment domains can enable an improved performance and data efficiency by exploiting structural biases in the class of environments. But environment class specialization has the drawback that the designed methods are most often not as tested as more general architectures which, owing to their widespread use and associated tuning, can often work out-of-the-box with much less implementation effort than an equivalent structured model. Currently, the most common type of architecture used in deep reinforcement learning for POMDPs is recurrent neural networks (RNNs) [135][53][102], typically the gated Long Short-Term Memory (LSTM) [89] variant which has far more stable optimization properties. Through a recurrent update mechanism, RNNs allow the efficient processing of sequential information into a fixed-length distributed vector that learns to represent summary statistics of the agent's history. This architecture's ubiquitous use is largely due to its generality as any sequence of input data can be processed by an RNN, and a large variety of open-source implementations of RL algorithms use an LSTM model as a default.

As an alternative to recurrent networks, a recent breakthrough development in deep sequence modeling is self-attention-based models [45, 164, 218, 207]. Self-attention bypasses several of the limitations of recurrent networks: enabling arbitrary indexing over time and no exploding / vanishing gradients [11][156]. The transformer [207], a particularly scalable and successful self-attention model, has become the de-facto state-of-the-art architecture for a large variety of natural language processing tasks [40, 164, 218, 207, 51, 126, 42, 218, 45, 164, 218, 42]. In this chapter, we propose a novel transformer architecture, the Gated Transformer-XL (GTrXL) [155], that remains as general as the LSTM in the types of RL domains it can be applied to. Extensive experimental validation demonstrates that our architectural variant of the transformer has improved optimization properties which allow it to (1) largely surpass LSTM's performance in difficult partially-observable and multitask meta-learning environments, (2) match or surpass LSTM's optimization stability. Our results are rigourously validated in a variety of challenging meta-learning domains including environments with hybrid action spaces (i.e. both discrete and continuous) and demonstrate its capability to perform meta-learning in a multitask setting, being performant even with a wide distribution of distinct environment classes.

Chapter 7 - Efficient Transformers in Reinforcement Learning using Actor-Learner Distillation [153]

As demonstrated in the previous chapter, the use of powerful and scalable general sequence models like transformers for reinforcement learning can potentially unlock a new level of artificial agent capabilities. However, training these sequence models require significant computational resources, even in supervised learning domains [83, 19]. These resource demands are further compounded when using reinforcement learning objectives, where data must be collected by stepping the policy observation-by-observation to collect new trajectories of experience to drive learning. Additionally, certain environments provide hard constraints on power and compute that limit the viable model complexity of Reinforcement Learning (RL) agents.

To work towards:

1. democratizing the use of transformer sequence models within the research community by reducing the computational cost of training such models, and,

2. unlocking the use of large model capacity while still operating within potential latency limits imposed by real systems during acting (e.g. robotic platforms),

in this chapter we develop the "Actor-Learner Distillation" (ALD) procedure. ALD leverages a continual form of distillation that transfers learning progress from a large capacity learner model to a small capacity actor model, considerably reducing computational cost by removing the requirement that the learner model has to be executed during environment interaction / data collection. With gated transformer models as the learner and LSTMs as the actor, we demonstrate in challenging memory and meta-learning environments that using Actor-Learner Distillation recovers the clear sample-efficiency gains of the gated transformer learner model, while still maintaining the fast inference and reduced total training time of the LSTM actor model.

1.3 Summary

In this thesis, we work towards the rapid generalization of artificial agents through the design of effective episodic memories. The contributions of this thesis are the following:

- In the first part of this thesis (covering Chapters 2 - 4), as an initial step towards effective, general meta-learners, we define an important class of environments, "Embodied Environments", and limit our focus to inter-episodic memories for meta-learning within this class. Embodied environments have agents with a physical manifestation in the environment resembling the natural world and represent an important class of environments for real-world applications, as most robotics applications are encompassed under this category. Furthermore, restricting our focus to these environments affords our algorithms a large amount of exploitable structure in state and observation spaces. We work towards the components that would construct a monolithic, end-to-end agent architecture for rapid adaptation in this environment class. The goal is an agent having the core competencies of memorization, planning and state inference.
  - In Chapter 2, we define the Neural Map, an inter-episodic memory that exploits geometric structure in embodied state spaces to efficiently map the physical world the agent inhabits into a spatially-organized feature map. Within a set of challenging embodied environments, we demonstrate significant improvements over more unstructured inter-episodic memories such as standard LSTMs.
  - In Chapter 3, we define architectural submodules that can be combined with and augment the basic capabilities of the Neural Map – namely, (1) a localization module that enables a rapid reduction in ambiguity over where the agent is located in an environment, given observations and a representation of a map of that environment, and (2) a planning module that allows rapid and generalizable path planning in map representations. We validate the performance of each submodule in isolation against related methods, and in both cases demonstrate a significant advancement in performance over previous techniques.
  - In Chapter 4, to address the missing capability of localization without a map, inspired by the traditional Simultaneous Localization and Mapping (SLAM) techniques, we describe a graph-based self-attention architecture that accomplishes both localization and mapping simultaneously. We present results demonstrating that such a model provides more accurate estimates of agent location compared to alternatives such as deep recurrent models and traditional SLAM implementations, even when tested in environments with significant amounts of perceptual aliasing.
- In the second part of the thesis, we move past our previous focus on embodied environments and present more general meta reinforcement learning methods which work across a target distribution of tasks encompassing any set of (PO)MDPs.
  - In Chapter 5, we describe a novel concurrent execution framework for meta-learning with inter-episodic memories. In previous work, an agent sequentially interacts with a target environment and performs *fast* adaptation conditioning on the sum total of past experience with that environment. This chapter describes an alternative execution paradigm, where multiple agents are executed in separate instances of the same environment in parallel and allowed to communicate to each other through a memory-based channel. The concurrent setting is especially important for applications where interaction is time-constrained, i.e. robotics or any environment which executes in real-time.
  - In Chapter 6, motivated by the substantial amount of success self-attention architectures have recently achieved in supervised learning, we present a gated transformer architecture designed specifically for improved optimization stability and performance in reinforcement learning tasks. The gated transformer is extensively validated across a challenging set of meta-learning environments, including a multitask setting. We show consistent and considerable gains over previous recurrent models, particularly whenever memory is a significant aspect of competancy in the environment.
  - In Chapter 7, we investigate the critical issue of computational cost for training transformer models in the (meta-)reinforcement learning setting. While an issue in supervised learning, inference speed greatly limits the amount of model capacity available for use

in reinforcement learning applications due to data collection requiring the step-wise execution of the transformer model. Large transformer models can take in excess of seconds to process states, especially on the un-accelerated hardware that distributed learning is typically performed on, meaning that in order to be feasibly trained, either (1) training would require an extreme number of parallel actors, or (2) the model's capacity must be inordinately constrained, hindering our ability to exploit the transformers impressive scaling properties. In this chapter, we propose Actor-Learner Distillation, a solution aiming to reduce the total computational cost of training transformers and democratizing their more widespread use. ALD is an online asymmetric distillation procedure that enables the transformer model to be only executed in a batched setting on accelerated hardware, while acting is done by more light-weight models better suited for fast inference.

- Finally, in Chapter 8 we conclude by summarizing the achievements of this thesis and discuss promising future avenues for meta-learining through inter-episodic memories.

Part I

Meta Reinforcement Learning in Embodied Environments

Chapter 2

Neural Map: Structured Inter-Episodic Memory

In the first half of this thesis, we present methods for meta-learning with inter-episodic memory that focus on a more specific class of meta-learning environments where the agent has a physical embodiment in its domain. These "embodied" environments have state, observation and dynamics structure that closely resemble that of the natural world, and for this reason they represent an important application area of meta-learning. In particular, within the class of embodied environments the common substructure manifests itself as (1) the laws of classical mechanics specify the transition dynamics, and (2) there exists latent 3d geometric structure inherent in all embodied observation- and state-spaces, regardless of the particular sensor modality. The majority of robotics applications can be encompassed under this category, and therefore methods effective here can be applied to wide areas of practical interest.

The strong regularities within embodied environments afford us substantial structure to make use of in the design of inter-episodic memories. Using this structure, in this chapter we present a memory map architecture that is spatially organized, intended to exploit the underlying 3D geometric composition of embodied environments. The "Neural Map" constructs a top-down representation of the 3D environment the agent is situated in in a particular meta-episode, endowing the agent with few-shot navigational capabilities by allowing it to quickly memorize the arrangement of novel surroundings. In more detail, the Neural Map is a 2-dimensional feature map where pixel positions in the map correspond to quantized positions in the environment. As an agent explores its 3D environment, the Neural Map writes the observation features to the nearest quantized position in the feature map corresponding to the agents real-world location. The agent policy then reads spatially local and global information from the stored map representation to plan its next action. To demonstrate the effectiveness of the Neural Map, we run it on a variety of challenging navigation-based meta-learning tasks in partially-observable environments with high-dimensional observations. We provide comparisons against more unstructured LSTM- and Memory Network-based models, and in all cases show a significant improvement over previous baselines.

2.1 Background & Related Work

Other than the straightforward architectures of combining an LSTM with Deep Reinforcement Learning (DRL) [135, 80], there has also been work on using more advanced external memory systems with DRL agents to handle partial observability. [146] used a memory network (MemNN) to solve maze-based environments similar to the ones presented in this paper. MemNN keeps the last M states in memory and encodes them into (key, value) feature pairs. It then queries this memory using a soft attention mechanism similar to the context operation of the Neural Map, except in the Neural Map the key/value features were written by the agent and aren't just a stored representation of the last M frames seen. [146] tested a few variants of this basic model, including ones which combined both LSTM and memory-network style memories.

In contrast to memory networks, another research direction is to design recurrent architectures that mimic computer memory systems. These architectures explicitly separate computation and memory in a way anagolous to a modern digital computer, in which some neural controller (akin to a CPU) interacts with an external memory (RAM). One recent model is similar to the Neural Map, called the Differentiable Neural Computer (DNC) [71], which combines a recurrent controller with an external memory system that allows several types of read/write access. In addition to defining an unconstrained write operator (in contrast to the neural map's write location being fixed), the DNC has a selective read operation that reads out the memory either by content or in the order that it was written. While the DNC is more specialized to solving algorithmic problems, the Neural Map can be seen as an extension of this Neural Computer framework to 3D environments, with a specific inductive bias on its write operator that allows sparse writes. Recently work has also been done toward sparsifying the read and write operations of the DNC [165]. This work was not focused on 3D environments and did not make any use of task-specific biases like agent location, but instead used more general biases like "Least-Recently-Used" memory addresses to force sparsity. More recently, the DNC, in conjunction with a VIN planning network [198], has been applied to the task of navigating partially-observable environments [107] although it still relied on supervised learning in order to train the complete system.

Contemporary to this work, [74] designed a similar 2D map structured memory to the Neural Map but which lacked a context addressing operation and used DAGGER [171], an imitation learning algorithm, to train their agent. Comparatively to their grid-like environments, Doom actions affect translational/rotational accelerations so training using imitation learning is more difficult since a search algorithm cannot be used directly as supervision. An interesting addition they made was the use of a multi-scale map representation and a Value Iteration network [198] to do better path planning. Later work expanded upon spatial-memory architectures, with the Neural SLAM architecture [221] extending spatial memories to settings where localization/odometry is not provided a priori, but instead has to be computed in tandem with the mapping of the environment. The MapNet [84] was an allocentric spatial memory that added a novel feature projection operator that mapped features to the points in the map that they are located in based on agent position and camera depth estimates. This is in contrast to the Neural Map, which maps all current observation features to the location they were *observed at* rather than their *external* environment position. They report better localization accuracy than traditional- and deep-learning-based baseline models on supervised learning datasets including a synthetic one based on Doom.

2.2 Neural Map

In this section, we will describe the details of the neural map. The neural map is the agent's internal memory storage that can be read from and written to during interaction with its environment, but where the write operator is selectively limited to affect only the part of the neural map that represents the area where the agent is currently located. For this paper, we assume for simplicity that we are dealing with a 2-dimensional feature map representation. This can easily be extended to 3-dimensional or even higher-dimensional maps (i.e. a 4D map with a 3D sub-map for each cardinal direction the agent can face).

Let the agent's position be (x, y) with $x \in \mathbb{R}$ and $y \in \mathbb{R}$ and let the neural map M be a $C \times H \times W$ feature block, where C is the feature dimension, H is the vertical extent of the map and W is the horizontal extent. Assume there exists some coordinate normalization function $\psi(x, y)$ such that every unique (x, y) can be mapped into (x', y'), where $x' \in \{0, \ldots, W-1\}$ and $y' \in \{0, \ldots, H-1\}$. For ease of notation, suppose in the sequel that all coordinates have been normalized by ψ into neural map space.

Let s_t be the current state embedding, M_t be the current neural map, and (x_t, y_t) be the current position of the agent. The Neural Map is defined by the following set of equations:

$$\begin{aligned}
r_t &= read(M_t),\\
c_t &= context(M_t, s_t, r_t),\\
w_{t+1}^{(x_t,y_t)} &= write(s_t, r_t, c_t, M_t^{(x_t,y_t)}),\\
M_{t+1} &= update(M_t, w_{t+1}^{(x_t,y_t)}),\\
o_t &= [r_t, c_t, w_{t+1}^{(x_t,y_t)}],\\
\pi_t(a|s) &= \text{Softmax}(f(o_t)),
\end{aligned} \tag{2.1}$$

where $w_t^{(x_t,y_t)}$ represents the feature at position (x_t, y_t) at time t, $[x_1, \ldots, x_k]$ represents a concatenation operation, and o_t is the output of the neural map at time t which is then processed by another deep network f to get the policy outputs $\pi_t(a|s)$. We will now separately describe each of the above operations in more detail:

Global Read Operation

The *read* operation passes the current neural map M_t through a deep convolutional network and produces a C-dimensional feature vector r_t. The global read vector r_t summarizes information about the entire map.

Context Read Operation:

The *context* operation performs context-based addressing to check whether certain features are stored in the map. It takes as input the current state embedding s_t and the current global read vector r_t and first produces a query vector q_t. The inner product of the query vector and each feature $M_t^{(x,y)}$ in the neural map is then taken to get scores $a_t^{(x,y)}$ at all positions (x, y). The scores are then normalized to get a probability distribution $\alpha_t^{(x,y)}$ over every position in the map, also known as "soft attention" [9]. This probability distribution is used to compute a weighted average c_t over all features $M_t^{(x,y)}$. To summarize:

$$\begin{aligned} q_t &= W[s_t, r_t], \\ a_t^{(x,y)} &= q_t \cdot M_t^{(x,y)}, \\ \alpha_t^{(x,y)} &= \frac{e^{a_t^{(x,y)}}}{\sum_{(w,z)} e^{a_t^{(w,z)}}}, \\ c_t &= \sum_{(x,y)} \alpha_t^{(x,y)} M_t^{(x,y)}, \end{aligned} \tag{2.2}$$

where W is a weight matrix. The context read operation allows the neural map to operate as an associative memory: the agent provides some possibly incomplete memory (the query vector q_t) and the operation will return the completed memory that most closely matches q_t. So, for example, the agent can query whether it has seen something similar to a particular landmark that is currently within its view.

Local Write Operation:

Given the agent's current position (x_t, y_t) at time t, the *write* operation takes as input the current state embedding s_t, the global read output r_t, the context read vector c_t and the current feature at position (x_t, y_t) in the neural map $M_t^{(x_t,y_t)}$ and produces, using a deep neural network f_w, a new C-dimensional vector $w_{t+1}^{(x_t,y_t)}$. This vector functions as the new local write candidate vector at the current position (x_t, y_t): $w_{t+1}^{(x_t,y_t)} = f_w([s_t, r_t, c_t, M_t^{(x_t,y_t)}])$

GRU-based Local Write Operation

As previously defined, the write operation simply replaces the vector at the agent's current position with a new feature produced by a deep network. Instead of this hard rewrite of the current position's feature vector, we can use a gated write operation based on the recurrent update equations of the Gated Recurrent Unit (GRU) [30]. Gated write operations have a long history in unstructured recurrent networks and they have shown a superior ability to maintain information over long time lags versus ungated networks. The GRU-based write operation is defined as:

$$\begin{aligned} r_{t+1}^{(x_t,y_t)} &= \sigma(W_r[s_t, r_t, c_t, M_t^{(x_t,y_t)}]) \\ \hat{w}_{t+1}^{(x_t,y_t)} &= \tanh(W_{\hat{h}}[s_t, r_t, c_t] + U_{\hat{h}}(r_{t+1}^{(x_t,y_t)} \odot M_t^{(x_t,y_t)})) \\ z_{t+1}^{(x_t,y_t)} &= \sigma(W_z[s_t, r_t, c_t, M_t^{(x_t,y_t)}]) \\ w_{t+1}^{(x_t,y_t)} &= (1 - z_{t+1}^{(x_t,y_t)}) \odot M_t^{(x_t,y_t)} + z_{t+1}^{(x_t,y_t)} \odot \hat{w}_{t+1}^{(x_t,y_t)}, \end{aligned} \tag{2.3}$$

where $x \odot y$ is the Hadamard product between vectors x and y, $\sigma(\cdot)$ is the sigmoid activation function and W_* and U_* are weight matrices. Using GRU terminology, $r_{t+1}^{(x_t,y_t)}$ is the reset gate,

$\hat{w}_{t+1}^{(x_t,y_t)}$ is the candidate activation and $z_{t+1}^{(x_t,y_t)}$ is the update gate. By making use of the reset and update gates, the GRU-based update can modulate how much the new write vector should differ from the currently stored feature.

Map Update Operation:

The *update* operation creates the neural map for the next time step. The new neural map M_{t+1} is equal to the old neural map M_t, except at the current agent position (x_t, y_t), where the current write candidate vector $w_{t+1}^{(x_t,y_t)}$ is stored:

$$M_{t+1}^{(a,b)} = \begin{cases} w_{t+1}^{(x_t,y_t)}, & \text{for } (a,b) = (x_t,y_t) \\ M_t^{(a,b)}, & \text{for } (a,b) \neq (x_t,y_t) \end{cases} \tag{2.4}$$

2.3 Ego-centric Neural Map

A major disadvantage of the neural map as previously described is that it requires some oracle to provide the current (x, y) position of the agent. This is a difficult problem in and of itself, and, despite being well studied, it is far from solved. The alternative to using absolute positions within the map is to use relative positions. That is, whenever the agent moves between time steps with some velocity (u, v), the map is counter-transformed by $(-u, -v)$, i.e. each feature in the map is shifted in the H and W dimensions. This will mean that the map will be ego-centric, i.e. the agent's position will stay stationary in the center of the neural map while the world as defined by the map moves around them. Therefore in this setup we only need some way of extracting the agent's velocity, which is typically a simpler task in real environments (for example, animals have inner ears and robots have accelerometers). Here we assume that there is some function $\xi(u', v')$ that discretizes the agent velocities (u', v') so that they represent valid velocities within the neural map (u, v). In the sequel, we assume that all velocies have been properly normalized by ξ into neural map space.

Let (pw, ph) be the center position of the neural map. The updated ego-centric neural map operations are shown below:

$$\begin{aligned} \overline{M}_t &= CounterTransform(M_t, (u_t, v_t)) \\ r_t &= read(\overline{M}_t) \\ c_t &= context(\overline{M}_t, s_t, r_t) \\ w_{t+1}^{(pw,ph)} &= write(s_t, r_t, c_t, \overline{M}_t^{(pw,ph)}) \\ M_{t+1} &= egoupdate(\overline{M}_t, w_{t+1}^{(pw,ph)}) \\ o_t &= [r_t, c_t, w_{t+1}^{(pw,ph)}] \\ \pi_t &= \text{Softmax}(f(o_t)) \end{aligned} \tag{2.5}$$

Where $\overline{M}_t$ is the current neural map M_t reverse transformed by the current velocity (u_t, v_t) so that the agents map position remains in the center (pw, ph).

Counter Transform Operation: The $CounterTransform$ operation transforms the current neural map M_t by the inverse of the agent's current velocity (u_t, v_t). Written formally:

$$\overline{M}_t^{(a,b)} = \begin{cases} M_t^{(a-u,b-v)}, & \text{for } {\scriptstyle (a-u)\in\{1,\ldots,W\}\wedge(b-v)\in\{1,\ldots,H\}} \\ 0, & \text{else} \end{cases} \tag{2.6}$$

While here we only deal with reverse translation, it is possible to handle rotations as well if the agent can measure it's angular velocity.

Map Egoupdate Operation: The *egoupdate* operation is functionally equivalent to the *update* operation except only the center position (pw, ph) is ever written to:

$$M_{t+1}^{(a,b)} = \begin{cases} w_{t+1}^{(pw,ph)}, & \text{for } (a,b) = (pw,ph) \\ \overline{M}_t^{(a,b)}, & \text{for } (a,b) \neq (pw,ph) \end{cases} \tag{2.7}$$

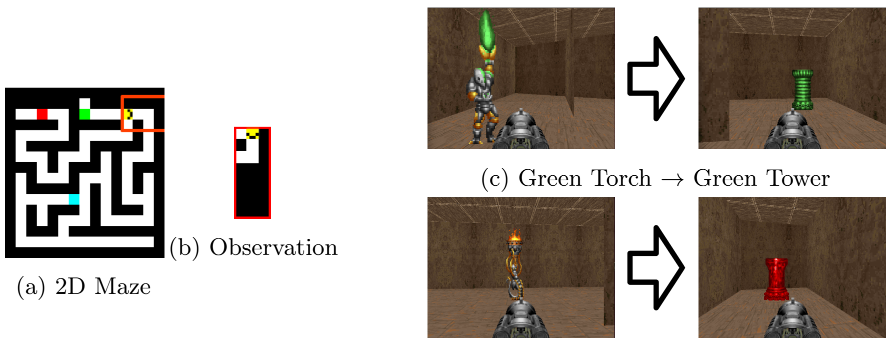

(a) 2D Maze (b) Observation (c) Green Torch $\rightarrow$ Green Tower (d) Red Torch $\rightarrow$ Red Tower

Figure 2.1: **Left:** Images showing the 2D maze environment. The left side (Fig. 2.1a) represents the fully observable maze while the right side (Fig. 2.1b) represents the agent observations. The agent is represented by the yellow pixel with its orientation indicated by the black arrow within the yellow block. The starting position is always the topmost position of the maze. The red bounding box represents the area of the maze that is subsampled for the agent observation. In "Goal-Search", the goal of the agent is to find a certain color block (either red or teal), where the correct color is provided by an indicator (either green or blue). This indicator has a fixed position near the start position of the agent. **Right:** State observations from the "Indicator" Doom maze environment. The agent starts in the middle of a maze looking in the direction of a torch indicator. The torch can be either green (top-left image) or red (bottom-left image) and indicates which of the goals to search for. The goals are two towers which are randomly located within the maze and match the indicator color. The episode ends whenever the agent touches a tower, whereupon it receives a positive reward if it reached the correct tower, while a negative reward otherwise.

2.4 Controller (Ego-)Neural Map

Here we describe the modification to the Neural Map we utilized for the 3D maze tasks. We include an extra state h that represents the hidden and cell state of an LSTM. The Neural Map equations are therefore:

$$\begin{aligned}
r_t &= read(M_t), \\
h_t &= LSTM(s_t, r_t, c_{t-1}, h_{t-1}), \\
c_t &= context(M_t, h_t), \\
w_{t+1}^{(x_t,y_t)} &= write(s_t, r_t, c_t, M_t^{(x_t,y_t)}), \\
M_{t+1} &= update(M_t, w_{t+1}^{(x_t,y_t)}), \\
o_t &= [r_t, c_t, w_{t+1}^{(x_t,y_t)}], \\
\pi_t(a|s) &= \text{Softmax}(f(o_t)),
\end{aligned} \tag{2.8}$$

2.5 Experiments

To demonstrate the effectiveness of the Neural Map, we run it on 2D and 3D maze-based meta-learning environments where memory is crucial to optimal behaviour. These meta-learning tasks differ in the semantics of objects between episodes and the maze layout, meaning each meta-episode interacts with a unique POMDP. We compare to previous memory-based deep RL agents, namely a LSTM-based agent as well as the Memory-Network-based MemNN [146] agents.

2.5.1 2D Goal-Search Environment

The "Goal-Search" environment is adapted from [146]. Here the agent starts in a fixed starting position within some randomly generated maze with two randomly positioned goal states. It then observes an indicator at a fixed position near the starting state (i.e. the green tile at the top of the maze in Fig. 2.1a). This indicator will tell the agent which of the two goals it needs to go to (blue indicator$\rightarrow$teal goal, green indicator$\rightarrow$red goal). If the agent goes to the correct goal, it gains a positive reward while if it goes to the incorrect goal it gains a negative reward. Therefore the agent needs to remember the indicator as it searches for the correct goal state. The mazes during training are generated using a random generator. A held-out set of 1000 random mazes

Agent	2D Goal-Search					
	Train			Test		
	7-11	13-15	Total	7-11	13-15	Total
Random	41.9%	25.7%	38.1%	46.0%	29.6%	38.8%
LSTM	84.7%	74.1%	87.4%	96.3%	83.4%	91.4%
MQN-32	80.2%	64.4%	83.3%	95.9%	74.6%	87.4%
MQN-64	83.2%	69.6%	85.8%	96.5%	76.7%	88.3%
Neural Map (15x15)	92.4%	80.5%	89.2%	93.5%	87.9%	91.7%
Neural Map + GRU (15x15)	97.0%	89.2%	94.9%	97.7%	94.0%	96.4%
Neural Map + GRU (8x8)	94.9%	90.7%	95.6%	98.0%	95.8%	97.3%
Neural Map + GRU + Pos (8x8)	95.0%	91.0%	95.9%	98.3%	94.3%	96.5%
Neural Map + GRU + Pos (6x6)	90.9%	83.2%	91.8%	97.1%	90.5%	94.0%
Ego Neural Map + GRU (15x15)	94.6%	91.1%	95.4%	97.7%	92.1%	95.5%
Ego Neural Map + GRU + Pos (15x15)	74.6%	63.9%	78.6%	87.8%	73.2%	82.7%

Table 2.1: Results of several different agent architectures on the “Goal-Search” environment. The “train” columns represents the number of mazes solved (in %) when sampling from the same distribution as used during training. The “test” columns represents the number of mazes solved when run on a set of held-out maze samples which are guaranteed not to have been sampled during training.

is kept for testing. This test set therefore represents maze geometries that have never been seen during training, and measure the agent’s ability to generalize to new environments. While this 2D benchmark is not of practical significance due to its simplicity, it enables more direct comparison to previously published results and also provides additional evidence of the Neural Map’s effectiveness at few-shot navigation in novel environments.

The first baseline agent we evaluate is a recurrent network with 128 LSTM units. The other baseline is the MQN, a previously published memory-network-based architecture that performs attention over the past K states it has seen [146]. Both LSTM and MQN models receive a one-hot encoding of the agent’s current location, previous velocity, and current orientation at each time step, in order to make the comparison to the fixed-frame Neural Map fair. We test these baselines against several Neural Map architectures, with each architecture having a different design choice.

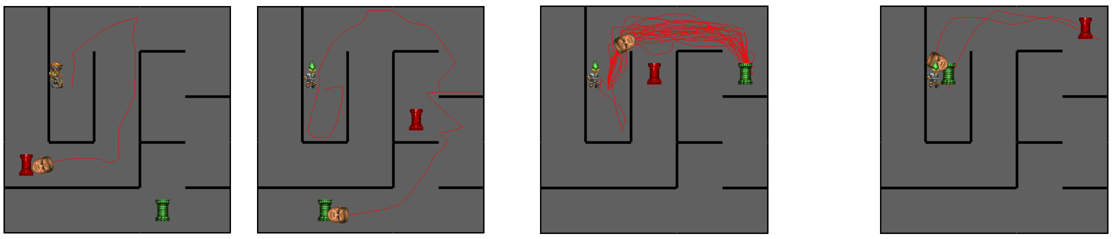
The results are reported in Table 2.1. During testing, we extend the maximum episode length from 100 to 500 steps so that the agent is given more time to solve the maze. The parentheses next to the model name represent the Neural Map dimensions of that particular model. From the results we can see that the Neural Map architectures solve the most mazes in both the training and test distributions compared to both LSTM and MQN baselines.

The results also demonstrate the effect of certain design decisions. One thing that can be observed is that using GRU updates adds several percentage points to the success rate (“Neural Map (15x15)” v.s. “Neural Map + GRU (15x15)”). We also tried downsampled Neural Maps, such that a pixel in the memory map represents several discrete locations in the environment. The Neural Map seems quite robust to this downsampling, with a downsampling of around 3 (6x6 v.s. 15x15) doing just a few percentage points worse, and still beating all baseline models. The 6x6 model has approximately the same number of memory cells as “MQN-32”, but its performance is much better, showing the benefit of having learnable write operations. For the egocentric model, in order to cover the entire map we set the pixels to be 2x smaller in each direction, so each pixel is only a quarter of a pixel in the fixed-frame map. Even with this coarser representation, the egocentric model did similarly to the fixed frame one. We demonstrate an example of what the Neural Map learned to address using its context operator in Section 2.5.5.

Finally, we tried adding the one-hot position encoding as a state input to the Neural Map, as is done for the baselines. We can see that there is a small improvement, but it is largely marginal, with the Neural Map doing a decent job of learning how to represent its own position without needing to be told explicitly. One interesting thing that we observed is that having the one-hot position encoding as an input to the egocentric map decreased performance, perhaps because it is difficult for the network to learn a mapping between fixed and egocentric frames.

Note that sometimes the percentage results are lower for the training distribution. This is mainly because the training set encompases almost all random mazes except the fixed 1000 of the test set, thus making it likely that the agent sees each training map only once.

Beyond train/test splits, the results are further separated by maze size. This information reveals that the memory networks are hardest hit by increasing maze size with sometimes a 20% drop in



(a) Indicator (b) Repeating (c) Minotaur

Figure 2.2: Top-down views showing succesful episodes in each of the 3 Doom maze tasks. The red lines indicate the path traveled by the agent. **Indicator** is shown in Fig. 2.2a, where the agent receives positive reward when entering the corresponding tower that matches the torch color it saw at the start of the episode and a negative reward otherwise. The episode terminates once the agent has reached a tower. **Repeating**, shown in Fig. 2.2b, has the same underlying mechanics except (1) the episode persists for T time steps regardless of towers entered and (2) the torch indicator is removed from the maze after the agent has reached a tower once. Therefore the agent needs to find the correct tower and then optimize its path to that tower. **Minotaur** shown in Fig. 2.2c requires the agent to reach the red goal and then return to the green goal that is at its starting position. Here the torch does not have any function. This fully-observable top-down view was not made available to the agent and is only used for visualization.

success on 13-15 v.s. 7-11. This is perhaps unsurprising given the inherent fixed time horizon of memory netwoks, and further reveals the benefit of using write-based memories.

2.5.2 3D Doom Environment Description

To demonstrate that our method can work in much more complicated 3D meta-learning environments with longer time horizons and high-dimensional image observations, we implemented three 3D maze environments using the ViZDoom [105] API and a random maze generator. Examples of all three environments are given in Figure 2.2.

Indicator Maze

The first environment is a recreation of the 2D indicator maze task, where an indicator is positioned in view of the player's starting state which is either a torch of red or green color. The goals are corresponding red/green towers that are randomly positioned throughout the maze that the player must locate.

Repeating Maze

The second environment is a variant of this indicator maze but whenever the player enters a goal state, it is teleported back to the beginning of the maze without terminating the episode (i.e. it retains its memory of the current maze). It gains a positive reward if it reaches the correct goal and a negative reward if it reaches the incorrect goal. After the first goal is reached, the correct indicator color is no longer displayed within the maze and a red indicator is displayed afterwards instead (regardless if the correct goal is green). An episode ends after a predetermined number of steps which depends on the maze size. The goal is therefore to find a path to the correct goal, and then optimize that path so that it can reach it as many times as possible.

Minotaur Maze

The third environment has the agent start in a fixed starting position next to the green tower, while the red tower is randomly placed somewhere in the maze. The agent receives a small positive reward if it reaches the red tower, and a larger positive reward if after reaching the red tower it returns to the green tower. Therefore the agent must efficiently navigate to the red goal while accurately remembering its entire path it so that it can backtrack to the start.

Agent		Indicator					Repeating					Minotaur				
Maze Size		4	5	6	7	8	4	5	6	7	8	4	5	6	7	8
LSTM	Acc	95.7	87.5	81.1	71.4	60.3	-	-	-	-	-	90.0	71.5	48.0	34.2	29.4
	Rew	-	-	-	-	-	7.26	7.58	6.06	5.32	4.98	1.35	1.07	0.72	0.51	0.44
FRMQN	Acc	87.3	82.9	78.0	72.0	59.8	-	-	-	-	-	72.7	54.5	38.8	28.8	23.7
	Rew	-	-	-	-	-	1.45	1.65	1.51	1.37	1.09	1.09	0.82	0.58	0.43	0.36
Controller	Acc	**95.8**	90.3	81.8	80.4	70.3	-	-	-	-	-	**99.7**	**92.2**	**67.5**	37.9	30.2
NMap	Rew	-	-	-	-	-	**17.4**	**17.1**	**12.0**	**11.4**	**12.3**	**1.50**	**1.38**	**1.01**	0.57	0.45
Controller	Acc	94.6	**91.0**	**87.6**	**85.8**	**72.2**	-	-	-	-	-	98.6	90.0	65.2	**44.7**	**33.8**
Ego-NMap	Rew	-	-	-	-	-	12.8	14.1	11.0	10.4	9.72	1.48	1.35	0.98	**0.67**	**0.51**

Table 2.2: Doom results on mazes not observed during training for the three tasks: Indicator, Repeating and Minotaur. Acc stands for Accuracy and Rew for Reward. Accuracy for Indicator means % of correct goals reached, while for Minotaur it means % of episodes where the agent successfully reached the goal and then backtracked to the beginning. Reward for Repeating is number of times correct goal was visited within the allotted time steps (+1 for correct goal, -1 for incorrect goal). Reward for Minotaur is +0.5 for reaching the goal and then +1.0 for backtracking to start after reaching goal (max episode reward is +1.5). We tested on maze sizes between [4,8] with 10 test mazes for each size. For each of the 50 total test mazes we ran 100 episodes with random goal locations and averaged the result.

2.5.3 3D Doom Model Description

All three environments used a RGB+D image of size 100x60 as input. We generate maze geometries randomly at train time but make sure to exclude a test set of 10 mazes for each size [4, 5, 6, 7, 8] (50 total). For these environments, we tested out four architectures:

Neural Map with Controller LSTM: Standard Neural Map with fixed frame addressing and GRU updates. We combine the neural map design with an LSTM that aggregates past state, read and context vectors and produces the query vector for the next time step's context read operation. See Section 2.4 for the modified Neural Map equations.

Ego Neural Map with Controller LSTM: Same as previous but with ego-centric addressing. The other difference is that the Ego Neural Map does not receive any positional input unlike the other 3 models, only receiving frame-by-frame ego-motion (quantized to a coarse grid).

LSTM: Single pre-output 256-dimensional LSTM layer.

FRMQN [146]: Memory network with LSTM feedback. This design uses an LSTM to make recurrent context queries to the memory network database. In addition, for the memory network baselines we did not set a fixed k but instead let it access any state from its entire episode. This means no information is lost to the memory network, it only needs to process its history.

The results are shown in Table 2.2. We can see that the Neural Map architectures work better than the baseline models, even though the memory network has access to its entire episode history at every time step. The ego-centric Neural Map beats the fixed frame map at Indicator, and gets similar performance on both Repeating and Minotaur environments, showing the ability of the Neural Map to function effectively even without global position information. It is possible that having a fixed frame makes path optimization easier, which would explain the larger rewards that the fixed-frame model got in the Repeating task. In the next section, we also investigated whether the neural map is robust to localization noise, which would be the case in a real world setting where we do not have access to a localization oracle and must instead rely on an error-prone odometry or SLAM-type algorithm to do localization.

For the baselines, we can see that FRMQN has difficulty learning on Repeating, only reaching the goal on average once. This could be because the indicator is only shown before the first goal is reached and so afterwards it needs to remember increasingly longer time horizons. Furthermore, because the red indicator is always shown after the first goal is reached, it might be difficult for the model to learn to do retrieval since the original correct indicator must be indexed by time and not image similarity. The FRMQN also has difficulty on Minotaur, probably due to needing to remember and organize a lot of spatial information (i.e. what actions were taken along the path). For Indicator, the FRMQN does similarly to the LSTM. We can see that the spatial structure of the Neural Map aids in optimizing the path in Repeating, averaging 12 goal reaches even in the largest maze size.

Agent		Indicator			Repeating			Minotaur		
Maze Size		4	5	6	4	5	6	4	5	6
$\sigma = 0$	Acc	88.4	84.4	79.3	-	-	-	97.3	89.0	62.0
	Rew	-	-	-	9.47	9.91	5.91	1.46	1.34	0.93
$\sigma = 0.01$	Acc	92.4	91.9	84.3	-	-	-	96.7	86.2	66.1
	Rew	-	-	-	12.4	12.9	10.9	1.45	1.29	0.99

Table 2.3: Results on the three 3D Doom maze tasks for the fixed-frame Neural Map with Controller LSTM. We can see that adding small compounding error does not largely affect the ability of the Neural Map to learn memory tasks and even has a beneficial effect for some tasks. Hyperparameters and architectures used were the same as presented in the main results.

2.5.4 Robustness to Odometric Noise

We present an experiment that featured drift noise to simulate the effects of the agent using a local visual odometry model that had small error in predicting each frame-by-frame transformation. This is meant to represent a more realistic scenario (e.g. robotic navigation) where perfect localization is not feasible but a relatively accurate estimate can be provided, demonstrating the robustness of the architecture to noise. For example, we could assume the Neural Map is run in parallel with a SLAM algorithm which provides an estimate of the agent's current position.

To model this noise, we add a zero-mean gaussian random variable to the oracle position with a variance that depends on the current time-step. In more detail, the noise-corrupted positions $(\hat{x}, \hat{y})$ in an $W \times W$ size map provided to the Neural Map are:

$$\begin{aligned}
&(\hat{x}, \hat{y}) = (\max\{\min\{\lfloor x + \epsilon_x \rfloor, W - 1\}, 0\}, \max\{\min\{\lfloor y + \epsilon_y \rfloor, W - 1\}, 0\}), \\
&\epsilon_x, \epsilon_y \sim \mathcal{N}(0, \sigma^2 t)
\end{aligned}$$

This simulates the effect of an odometry algorithm which has independent zero-mean gaussian error with equal variance. This error compounds over time causing the variance to grow with the time step. We evaluate the Neural Map with noise $\sigma = 1/100$ on smaller versions of the 3D Doom maze tasks (maze sizes [4, 5, 6]) and compare it to the version with perfect odometry. We train for 1500 steps of 100 episodes each step. Results are shown in Table 2.3. We can see that adding a small amount of error at each time step does not largely affect the results of the memory and can even benefit it, with some noticeable improvements on Indicator and Repeating tasks. It's possible that the noise acts as a regularizer to speed up learning. For the Minotaur task, since positional information is important because the agent must remember the entire path taken, adding noise causes a slight decrease in reward in mazes of size 4 and 5, but otherwise performance is very similar. These results provide evidence that the Neural Map is likely to work in the case where a localization oracle is not available and instead only error-prone odometry is.

We also plot some example trajectories to compare the effect of noise. We can see that the noise causes some slight aliasing in the position, which increases as time passes. The positions are quantized to a 15×15 grid.

2.5.5 Samples of Context Read Distribution

2D Environment

To provide some insight into what the Neural Map learns, we show samples of the probability distribution given by the context read operation in a 2D maze example. We ran it on an example maze shown in Figure 2.4. In this figure, the top row of images are the agent observations, the center row are the fully observable mazes and the bottom row are the probability distributions over locations from the context operation, e.g. the $\alpha_t^{(x,y)}$ values defined by Eq. 2.2. In this maze, the indicator is blue, which indicates that the teal goal should be visited. We can see that once the agent sees the incorrect red goal, the context distribution faintly focuses on the map location where the agent had observed the indicator. On the other hand, when the agent first observes the correct teal goal, the location where the agent observed the indicator lights up brightly. This means that the agent is using its context retrieval operation to keep track of the landmark (the indicator) that it has previously seen.

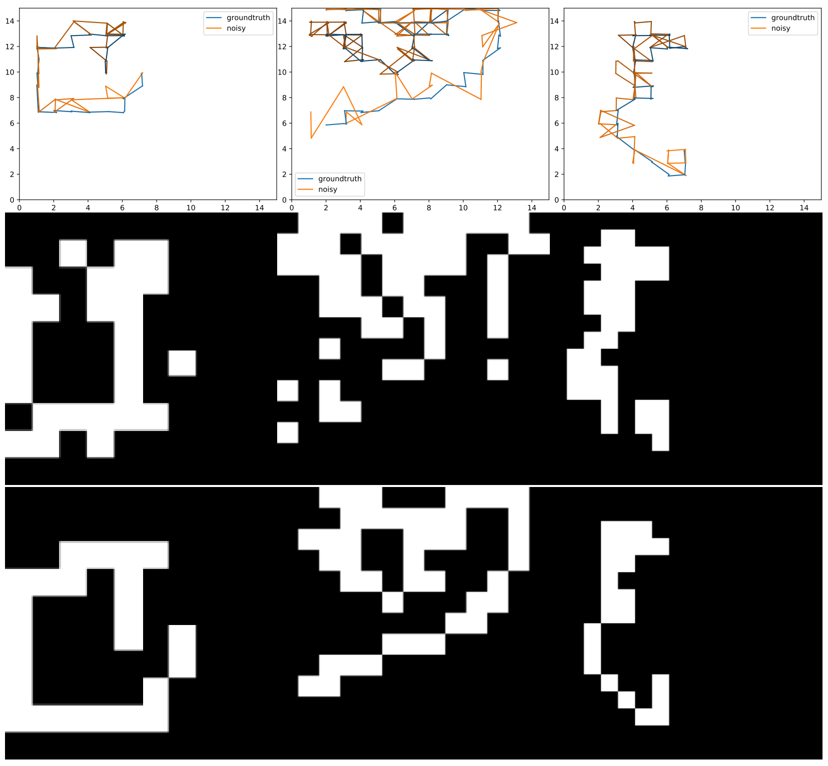


Figure 2.3: **Top:** Noisy v.s. Groundtruth Position trajectory (quantized to a 15×15 grid). As time progresses, the colors get lighter. **Center:** Neural Map cells addressed by the write operator under the noisy positions. **Bottom:** Neural Map cells that would have been written to under perfect position estimates.

3D Environment

We draw some examples of the context addressing probability distribution in the 3D Doom environment in Figure 2.5 for the allocentric Neural Map We can see that the Neural Map learns to use its context addressing operator to retrieve the indicator torch identity, until it sees the correct corresponding tower. Once it sees the correct tower there is a shift in how the agent uses the map and the probability map seems to invert, addressing the parts of the map that were unexplored. This effect was found to be consistent in both allocentric and egocentric variants. This might be because the Neural Map variant used on Doom had an internal LSTM which could enable it to remember the indicator identity for the short amount of time it took to walk up to the goal.

2.5.6 Backtracking

We also explored whether the allocentric and egocentric Neural Maps were capable of using their memories in order to do backtracking, i.e. re-visiting unexplored areas of the maze. To measure this, we developed a variant of the Indicator Maze where the goal states were removed. We want to measure how much of the maze is explored by the agent under this setting where there are no terminal states. To measure how much of the maze was explored, we quantized the 50 test mazes into 11 discrete positions and counted how many of the quantized positions the agent visited. We report results below in Table 2.4

2.6 Discussion

Navigation is a core competency for all embodied applications, as being able to remember and revisit parts of the environment is crucial for any successful behaviour. In this chapter we described an inter-episodic memory architecture, the Neural Map, which accomplishes few-shot navigation by organizing its memory into the spatial structure of a 2D map (generalizable to 3D and higher dimensions). The map structure enables sparse writes to the memory, where the address of the

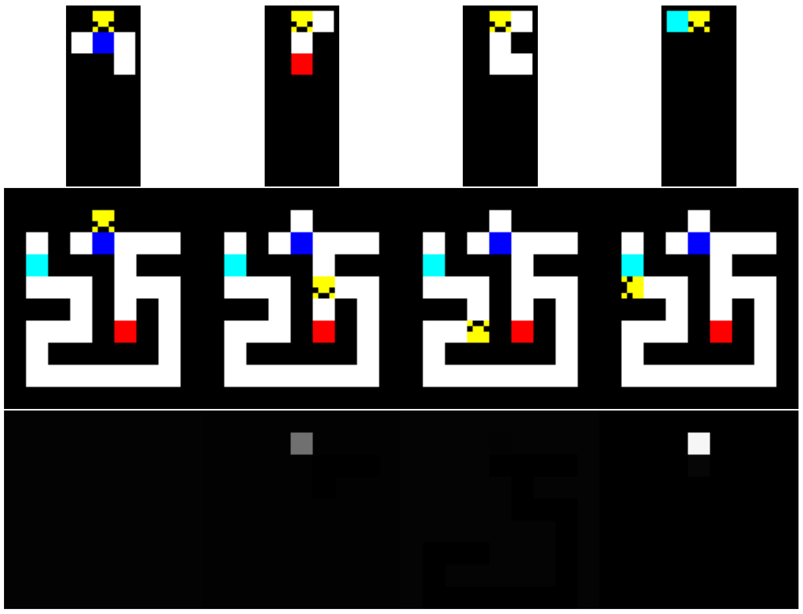


Figure 2.4: A few sampled states from an example episode demonstrating how the agent learns to use the context addressing operation of the Neural Map. The top row of images is the observations made by the agent, the center is the fully observable mazes and the bottom image is the probability distributions over locations induced by the context operation at that step.

Agent	Visitation Score
Controller NMap	71.6%
Controller Ego-NMap	77.6%
LSTM	68.5%

Table 2.4: Visitation scores of the Neural Map models which measure how much of a maze is explored within a set time limit. We can see that the egocentric neural map explores more of the mazes than the allocentric model, exploring on average 77.6% of the test mazes. The allocentric neural map explores 71.6% of the test mazes. The LSTM is reported to provide a point of comparison.

write is in a correspondence to the agent's current position in the environment. We showed the Neural Map's ability to learn, directly using reinforcement learning objectives, how to behave within challenging 2D and 3D few-shot navigation tasks that required storing, accessing and composing information over long time horizons. The results demonstrated that our architecture surpassed baseline memories used in previous work by significant margins. Additionally, we ablated the benefit of certain design decisions made in our architecture: using GRU updates instead of hard writes, demonstrating that the ego-centric viewpoint does not diminish performance, that the Neural Map is robust to downsampling its memory and that it is robust to odometric noise. To show that our method can scale up to difficult 3D environments, we implemented several new maze environments in Doom and demonstrated that a hybrid Neural Map + LSTM model solves the scenarios at a performance higher than previous DRL memory-based architectures.

Beyond the Neural Map's ability to rapidly memorize an environment, there exist other core competancies in embodied environments that can improve the adaptation speed of meta-learning agents. These other competencies include planning, which enables the agent to look ahead into potential futures to decide a more optimal sequence of actions, and state inference, which can help an agent disambiguate its state given a map of its environments. In the next chapter, towards our goal of a monolithic agent architecture we visit these other capabilities and, by exploiting the biases inherent to embodied environments, we design submodules which can augment the Neural Map to perform efficient and effective planning and state inference. We validate each of these methods in isolation against previous baselines and demonstrate large improvements, reaching state-of-the-art performance.

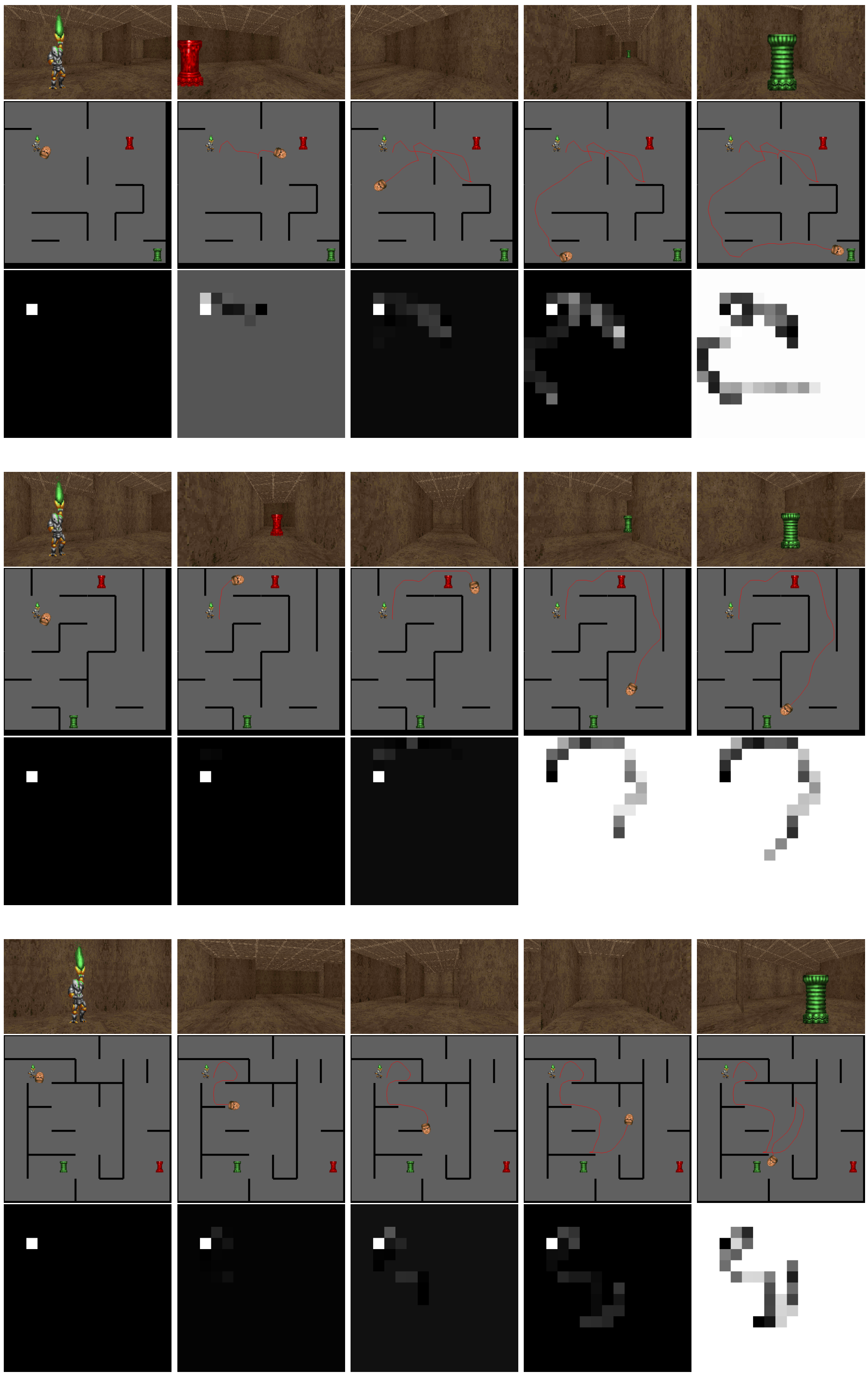


Figure 2.5: Three example episodes of the (allocentric) context addressing operator on Doom mazes. The top images of each row are the RGB inputs the agent sees, the center images are a top-down representation of the maze, and the bottom images are the $\alpha_t^{(x,y)}$ of the context operation.

Chapter 3

Planning and State Inference in Embodied Inter-Episodic Memories

In the previous chapter, we defined a spatially-structured inter-episodic memory designed specifically for few-shot navigation in embodied environments. By exploiting the geometric composition of the environment, the Neural Map enabled faster memorization of the unique environments seen in each meta-episode and, correspondingly, faster adaptation within the task instance. Despite the Neural Map's capacity for rapid memorization in embodied environments, there is yet still more structure that can be exploited beyond just the agent recalling specifics about previous observations. In particular, by providing priors through architecture on the computation being performed on the memory map representation, we can extend the Neural Map to be competent on two additional, important capabilities: namely, planning and state inference.

All planning in embodied environments is a form of "path planning" where, given a goal location, an agent desires a shortest path through the environment to get to that location. Our first augmentation to the memory map representation previously described is that of an architecture designed to do efficient, generalizable and effective path-planning, the Gated Path Planning Network (GPPN). The GPPN builds off previous work [199] which designed a differentiable sub-module that performs path-finding as directed by the agent in some inner loop. These *Value Iteration Network (VIN)* modules mimic the application of Value Iteration on a 2D grid world, but without a pre-specified model or reward function. VINs were shown to be capable of computing near-optimal paths in 2D mazes and 3D landscapes where the transition model $P(s'|s,a)$ was not provided a priori and had to be learned. In this chapter, we will show that VINs are often plagued by training instability, oscillating between high and low performance between epochs; random seed sensitivity, often converging to different performances depending on the random seed that was used; and hyperparameter sensitivity, where relatively small changes in hyperparameters can cause diverging behaviour. Owing to these optimization difficulties, we reframe the VIN as a recurrent-convolutional network, which enables us to replace the unconventional recurrent VIN update (convolution & max-pooling) with well-established gated recurrent operators such as the LSTM update [89]. This results in the *Gated Path Planning Networks (GPPNs)*, a more general model that relaxes the architectural inductive bias of VINs. The resultant architecture is thus no longer constrained to perform a computation resembling value-iteration.

An equally important capability in embodied environments, state inference, consits of finding an agents location given a map representation of the current surroundings and a stream of agent observations. Localization, the ability to localize under uncertainty, is required by autonomous agents to perform various downstream tasks such as planning, exploration and target-navigation, and is required when our agents *revisit* environments that were already previously mapped. In this chapter, we propose another architecture, the Active Neural Localizer, suited to tackle the global localization problem where the initial position of the agent is unknown. Localization remains an open problem, and there are not many methods developed which can be learnt from data in an end-to-end manner, an aspect that is required when using memory maps where the particular semantics of stored feature representations are opaque to the agent designer. Instead, previous traditional localization methods relied on lengthy hand-tuning and feature selection by domain experts. Additionally, the vast majority of previous localization approaches were *passive*, meaning that they passively estimate the position of the agent from the stream of incoming observations, and do not have the ability to decide the actions taken by the agent. The ability to decide the actions can result in faster as well as more accurate localization as the agent can learn to navigate quickly to unambiguous locations in the environment. Unlike previous methods, the proposed Active Neural Localizer is both (1) end-to-end differentiable, enabling the combination of such an architecture with Neural-Map-like memories, and (2) active, choosing actions to reduce uncertainty as quickly as possible. Based on the Bayesian filtering algorithm for localization [65], the Active Neural Localizer contains a perceptual model to estimate the likelihood of the agent's observations, a structured component for representing the belief, multiplicative interactions to propagate the belief based on observations and a policy model over the current belief to localize accurately while minimizing the number of steps required for localization.

In order to ablate against confounding effects, which could be introduced in the case of a monolithic architecture, the chapter presents empirical results that test each sub-module, the GPPN and the Active Neural Localizer, in isolation against previous baselines but on the exact same set of environment domains. For GPPNs, we first empirically establish that they perform better or equal to the performance of VINs on a wide variety of 2D maze experiments, including different transition models, maze sizes and different training dataset sizes. We further demonstrate that GPNNs exhibit fewer optimization issues than VINs, including reducing random seed and hyperparameter sensitivity and increasing training stability. GPPNs are also shown to work with larger kernel sizes, often outperforming VINs with significantly fewer recurrent iterations, and also learn faster on average and generalize better given less training samples. Finally, we present results for both VIN and GPPN on challenging 3D ViZDoom environments [105], where the planner is only provided with first-person RGB images instead of the top-down 2D maze design. Additionally, we present results that demonstrate the Active Neural Localizer is capable of rapid localization, can capably choose actions to reduce uncertainty and, finally, can generalize not only to unseen test maps in the same class of environments but also across completely separate domains.

3.1 Background & Related Work

There is a rapidly emerging literature surrounding differentiable submodules that can perform either state inference or planning due to its highly desirable practical ability to avoid lengthy and expensive model specification by an agent-designer, instead learning MDP parameters necessary for planning directly through data. For planning, [103] looked at extending differentiable planning towards being able to plan in partially observable environments. In their setting, the agent is not provided a-priori with its position within the environment and thus needs to maintain a belief state over where it actually is. Similar to VIN's differentiable extension of VI, the QMDP-Net architecture was based on creating a differentiable analogue of the QMDP algorithm [123], an algorithm designed to approximate belief space planning in POMDPs. The architecture itself consisted of a filter module, which maintained the beliefs over which states the agent currently was in, and a planning module, which determined what action to take next. The planning module was essentially using a VIN to enable it to make more informed decisions on which parts of the environment to explore.

In recent work there has been a variety of deep reinforcement learning models that have examined combining an internal planning process with model-free methods. The Predictron [186] was a value function approximator which predicted a policy's value by internally rolling out an LSTM forward predictive model of the agent's future rewards, discounts and values. These future rewards, values and discounts were then accumulated together, with the idea that this would predict a more accurate value by forcing the architecture to model a multi-step rollout. A later extension, Value Predictive Networks [147], learnt a forward model that is used to predict the future rewards and values of executing a multi-step rollout. Although similar to the Predictron, they considered the control setting, where not only a value function had to be learnt but a policy as well. They demonstrated that their model, trained using model-free methods, was able to outperform existing methods on a 2D goal navigation task and outperformed DQN on Atari games.

Recent work has also made progress towards end-to-end localization using deep learning models. [133] showed that a stacked LSTM can do reasonably well at self-localization. The model consisted of a deep convolutional network which took in at each time step state observations, reward, agent velocity and previous actions. To improve performance, the model also used several auxiliary objectives such as depth prediction and loop closure detection. The agent was successful at navigation tasks within complex 3D mazes. Additionally, the hidden states learned by the models were shown to be quite accurate at predicting agent position, even though the LSTM was not explicitly trained to do so. Other works have looked at doing end-to-end relocalization more explicitly. One such method, called PoseNet [106], used a deep convolutional network to implicitly represent the scene, mapping a single monocular image to a 3D pose (position and orientation). This method is limited by the fact that it requires a new PoseNet trained on each scene since the map is represented implicitly by the convnet weights, and is unable to transfer to scenes not observed during training. An extension to PoseNet, called VidLoc [33], utilized temporal information to make more accurate estimates of the poses by passing a Bidirectional LSTM over each monocular image in a sequence, enabling a trainable smoothing filter over the pose estimates. Both these methods lack a straightforward method to utilize past map data to do localization in a new environment. In contrast, we demonstrate our method is capable of generalizing to new maps that were not previously seen during training time.

Localization in general has been an active field of research since more than two decades. In the context of mobile autonomous agents, localization can refer to two broad classes of problems: Local localization and Global localization. Local localization methods assume that the initial position of the agent is known and they aim to track the position as it moves. A large number of localization methods tackle only the problem of local localization. These include classical methods based on Kalman Filters [100, 187] geometry-based visual odometry methods [145] and most recently, learning-based visual odometry methods which learn to predict motion between consecutive frames using recurrent convolutional neural networks [34, 211]. Local localization techniques often make restrictive assumptions about the agent's location. Kalman filters assume Gaussian distributed initial uncertainty, while the visual odometry-based methods only predict the relative motion between consecutive frames or with respect to the initial frame using camera images. Consequently, they are unable to tackle the global localization problem where the initial position of the agent is unknown. This also results in their inability to handle localization failures, which consequently leads to the requirement of constant human monitoring and intervention [21].

Global localization is more challenging than the local localization problem and is also considered as the basic precondition for truly autonomous agents by [21]. Among the methods for global localization, the proposed method in this chapter is closest to Markov Localization [63]. In contrast to local localization approaches, Markov Localization computes a probability distribution over all the possible locations in the environment. This probability distribution, also known as the *belief*, is represented using piecewise constant functions (or histograms) over the state space and propagated using the Markov assumption. Other methods for global localization include Multi-hypothesis Kalman filters [37, 172] which use a mixture of Gaussians to represent the belief and Monte Carlo Localization [203] which use a set of samples (or *particles*) to represent the belief.

All the above localization methods are *passive*, meaning that they aren't capable of deciding the actions to localize more accurately and efficiently. There has been very little research on *active* localization approaches. Active Markov Localization [64] is the active variant of Markov Localization where the agent chooses actions greedily to maximize the reduction in the entropy of the belief. [96] presented the active variant of Multi-hypothesis Kalman filters where actions are chosen to optimise the information gathering for localization. Both of these methods do not learn from data and have very high computational complexity. In contrast, we demonstrate that the proposed method is several order of magnitudes faster while being more accurate and is capable of learning from data and generalizing well to unseen environments.

3.2 Environments and Maze Transition Types

We test both Gated Path Planning Networks, Active Neural Localizer and baselines on 2D maze environments and 3D ViZDoom environments. For the experiments, we vary the datasets along a variety of axes such as training dataset size, maze size and maze transition kernel. Additionally for Active Neural Localizer, we use a high-fidelity office environment based on the Unreal3D engine to demonstrate the transferability of the localizer submodule, which achieves high zero-shot localization accuracy on the Unreal3D environment. even when trained only on 3D ViZDoom mazes. We next describe each of the MDP classes in more detail.

We describe notation used in the sequel here. Let t be the current episode step. Let y_t be a random variable denoting the state, or location, of the agent, represented as a tuple A_x, A_y where A_x,A_y denote the agent's x-coordinate, y-coordinate and orientation respectively. Let $M \times N$ be the map size. Then, $A_x \in [1, M]$, $A_y \in [1, N]$.

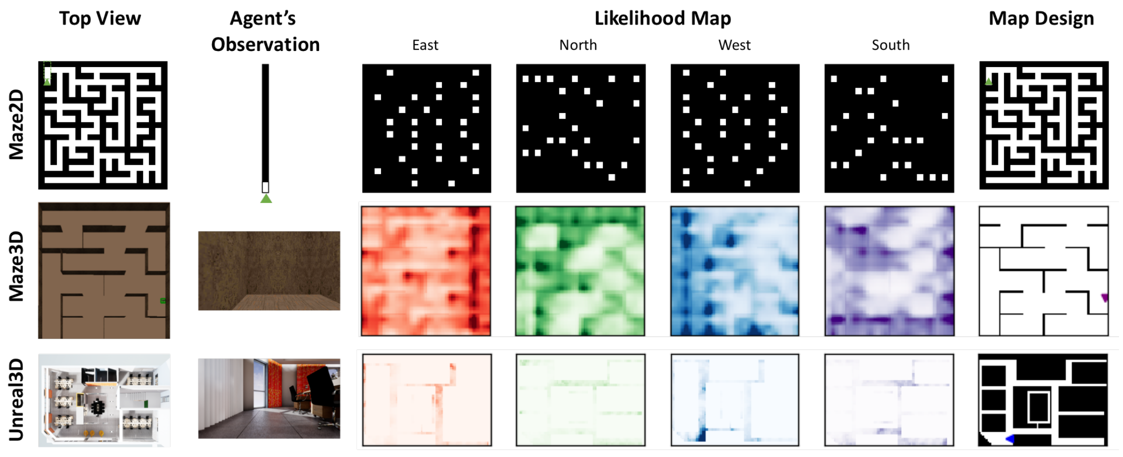


Figure 3.1: The map design, agent's observation and the corresponding likelihood maps in different domains. In 2D domains, agent's observation is the pixels in front of the agent until the first obstacle. In the 3D domain, the agent's observation is the image showing the first-person view of the world as seen by the agent.

3.2.1 Domain Descriptions

2D Mazes

The 2D maze environment is created with a maze generation process that uses either Kruskal's algorithm or Depth-First Search with the Recursive Backtracker algorithm [129] to construct the maze tree, always resulting in fully connected mazes. For each maze, we sample a probability d uniformly from [0,1]. Then for each wall, we delete the wall with probability d. 2D maze MDPs are represented by a binary matrix, where 0 denotes an obstacle and 1 denotes free space. This binary matrix, referred to as the "Map Design", is available as input to both localization and planning submodules (see Figure 3.1, top-left, for an example of the maze design).

For localization, we simulate partial-observability by constructing the agent's observation as the series of pixels in front of the agent. For a map size of $M \times N$, the agent's observation is an array of size $max(M, N)$ containing the pixel values in front of the agent. The view of the agent is obscured by obstacles, so all pixel values behind the first obstacle are treated as 0. The top row in Figure 3.1 shows examples of map design and agent observation in this environment. The experiments in the 2D environments are designed to evaluate and quantify the effectiveness of the submodules in ideal conditions, without observation or motion noise, and the size of the 2D environments can also be easily varied to test scalability. This environment also provides a benchmark similar to previous published results, such as in [198] and [104].

3D ViZDoom Mazes

We use the Doom Game Engine and the ViZDoom API [105] to create mazes in a simulated 3D environment. The maze design for the 3D mazes are generated in exactly the same manner as the 2D mazes, using either Kruskal's algorithm or Depth-First Search with the Recursive Backtracker algorithm followed by wall pruning with a uniformly sampled probability d. To construct the map design for each Doom maze, we take RGB screenshots showing the first-person view of the environment at each position on a regularly-spaced grid and the 4 cardinal orientations (North, East, West, South). A sample 3D Doom maze and example screenshot images are shown in the second row of Figure 3.1. For an $M \times N$ maze and with the 4 orientations, the map design consists of a total of $4MN$ images. In the 3D ViZDoom experiments, the map images obtained on the grid are given as input to the model (instead of the 2D map design). These images are featurized and resemble the distributed representations potentially produced by the Neural Map or a related memory map architecture.

Unreal3D Office

Unreal3D is a photo-realistic simulation environment built using the Unreal Game Engine. We use the AirSim API [183] to interact with the game engine. The environment consists of a modern office space as shown in the third row of Figure 3.1 obtained from the Unreal Engine Marketplace[1]. This environment was only used to demonstrate the transferability of the Active Neural Localizer.

3.2.2 Transition Types

We define here different maze transition kernels: In **NEWS**, the agent can move North, East, West, or South; in **Differential Drive**, the agent can move forward along its current orientation, or turn left/right by 90 degrees; in **Moore**, the agent can move to any of the eight cells in its Moore neighborhood. When defining a goal for the planning sub-module to plan towards, the dimension of the goal map given as input to the model is $1 \times M \times N$ for NEWS and Moore, and $4 \times M \times N$ for Differential Drive (which has an cardinally oriented goal location). For localization tasks, we only consider the oriented "Differential Drive" transition kernel due to its more realistic field-of-view, which more closely resembles the field-of-view in real-world sensor modalities.

3.3 Gated Path Planning Networks

Given a Markov decision process (MDP) consisting of states s, actions a, a reward function $\mathcal{R}$, and state transition kernels $P(s' \mid s, a)$, we recall here that *Value Iteration* is a method of computing an optimal policy π and its value $V^\pi(s) = \mathbb{E}^\pi\left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t, s_{t+1}) \mid s_0 = s\right]$, where $\gamma \in [0, 1]$ is a discount factor and $\mathcal{R}(s_t, a_t, s_{t+1})$ is a reward function. More specifically, value iteration starts with an arbitrary function $V^{(0)}$ and iteratively computes:

$$\begin{aligned} Q^{(k)}(s,a) &= \sum_{s'} P(s' \mid s, a)\left(\mathcal{R}(s,a,s') + \gamma V^{(k-1)}(s')\right), \\ V^{(k)}(s) &= \max_a Q^{(k)}(s,a). \end{aligned}$$

[199] introduced the *Value Iteration Network (VIN)*, capable of learning MDP parameters in grid-based environments from data automatically. The VIN reformulates value iteration as a recursive process of applying convolutions and max-pooling over the feature channels:

$$\begin{aligned} \bar{Q}^{(k)}_{\bar{a},i',j'} &= \sum_{i,j}\left(W^R_{\bar{a},i,j}\bar{R}_{i'-i,j'-j} + W^V_{\bar{a},i,j}\bar{V}^{(k-1)}_{i'-i,j'-j}\right), \\ \bar{V}^{(k)}_{i,j} &= \max_{\bar{a}} \bar{Q}^{(k)}_{\bar{a},i,j}, \end{aligned} \tag{3.1}$$

where the indices $i, j \in [m]$ correspond to cells in the $m \times m$ maze, $\bar{R}, \bar{Q}, \bar{V}$ is the VIN estimated reward, action-value and value functions, respectively, $\bar{a}$ is the action index of the $\bar{Q}$ feature map, and W^R, W^V are the convolutional weights for the reward function and value function, respectively. In the following iteration, the previous value $\bar{V}$ is stacked with $\bar{R}$ for the convolution step. While constrained to MDPs with grid-based structure, this covers the embodied environments we consider in this part of the thesis. [199] showed that VINs have much greater success at path planning than baseline CNN and feedforward architectures in a variety of 2D and graph-based navigation tasks. The demonstrated success of VIN has made it an important component of models designed to solve downstream tasks where navigation is crucial [103, 75, 77].

We now describe the Gated Path Planning Network and then comprehensively evaluate its performance compared to the previous state-of-the-art methods for differentiable path planning. The design of the GPPN is motivated by asking whether the inductive biases provided by the original differentiable path planning algorithm, the VIN, are even necessary: is it possible that using alternative, more general architectures might work better than those of the VIN? We can view the VIN update (3.1) within the perspective of a convolutional-recurrent network, updating a

[1] `https://www.unrealengine.com/marketplace/small-office-prop-pack`

recurrent state $V^{(k)}_{i',j'}$ at every spatial position (i',j') in each iteration:

$$\begin{aligned}\bar{V}^{(k)}_{i',j'} &= \max_{\bar{a}} \left(\sum_{i,j} W^R_{\bar{a},i,j} \bar{R}_{i'-i,j'-j} + W^V_{\bar{a},i,j} \bar{V}^{(k-1)}_{i'-i,j'-j} \right) \\ &= \max_{\bar{a}} \left(W^R_{\bar{a}} \bar{R}_{[i',j',3]} + W^V_{\bar{a}} \bar{V}^{(k-1)}_{[i',j',3]} \right), \end{aligned} \tag{3.2}$$

where $X_{[i',j',F]}$ denotes the image patch centered at position (i',j') with kernel size F. From (3.2), it can be seen that VIN follows the standard recurrent neural network (RNN) update where the recurrent state is updated by taking a linear combination of the input $\bar{R}$ and the previous recurrent state $\bar{V}^{(k-1)}$, and passing their sum through a nonlinearity $\max_{\bar{a}}$. The main differences from a standard RNN are the following: the unconventional nonlinearity (channel-wise max-pooling) used in VIN; the hidden dimension of the recurrent network, which is essentially one; the sparse weight matrices, where the non-zero values of the weight matrices represent neighboring inputs and units which are local in space; and the restriction of kernel sizes to 3.

Under this perspective, it is easy to question whether the adherence to these strict architectural biases is even necessary, given the long history of demonstrations that standard non-gated recurrent operators are difficult to optimize due to effects such as vanishing and exploding gradients [156].

We can easily replace the recurrent VIN update in (3.2) with the well-established LSTM update [89], whose gated update alleviates many of the problems with standard recurrent networks:

$$h^{(k)}_{i',j'}, c^{(k)}_{i',j'} = \mathbf{LSTM} \left(\sum_{\bar{a}} \left(W^R_{\bar{a}} \bar{R}_{[i',j',F]} + W^h_{\bar{a}} h^{(k-1)}_{[i',j',F]} \right), c^{(k-1)}_{i',j'} \right), \tag{3.3}$$

where F is the convolution kernel size. This recurrent update (3.3) still maintains the convolutional properties of the input and recurrent weight matrix as in VIN. It involves taking as input the $F \times F$ convolution of the input vector $\bar{R}$ and previous hidden states $h^{(k-1)}$, and the previous cell state $c^{(k-1)}_{i',j'}$ of the LSTM at the central position (i',j'). We call path planning modules which use these gated updates *Gated Path Planning Networks (GPPNs)*. The GPPN is an LSTM which uses convolution of previous spatially-contiguous hidden states for its input.

3.3.1 Planning Experiments

We now empirically compare VIN and GPPN using two metrics: **%Optimal** (%Opt) is the percentage of states whose predicted paths under the policy estimated by the model has optimal length, and **%Success** (%Suc) is the percentage of states whose predicted paths under the policy estimated by the model reach the goal state. The reported performance is on a held-out test split. In contrast with the metrics reported in [199], we do not stochastically sample rollouts but instead evaluate and train the output policy of the models directly on all states simultaneously. This reduces optimization noise and makes it easier to tell whether difficulties with training are due to sampling noise or model architecture/capacity.

All analyses are based on 2D maze results, except in Section 3.3.1 where we discuss 3D ViZDoom results. In order to make comparison fair, we utilized a hidden dimension of 150 for GPPN and 600 for VIN, owing to the approximately 4× increase in parameters a GPPN contains due to the 4 gates it computes. Unless otherwise noted, the results were obtained by doing a hyperparameter sweep of (K,F) over $K \in \{5,10,15,20,30\}$ and $F \in \{3,5,7,9,11\}$, and using a 25k/5k/5k train-val-test split. Other experimental details are deferred to the Appendix.

Varying Kernel Size F

One question that can be asked of the architectural choices of the VIN is whether the kernel size needs to be the same dimension as the true underlying transition model. The kernel size used in VIN was set to 3×3 with a stride of 1, which is sufficient to represent the true transition model when the agent can move anywhere in the Moore neighborhood, but it limits the rate at which information propagates spatially with each iteration. With a kernel size of 3×3 and stride of 1, the receptive field of a unit in the last iteration's feature map increases with rate $(3 + 2K) \times (3 + 2K)$ where K is the iteration count, meaning that the maximum path length information travels scales directly with iteration count k. Therefore for long-term planning in larger environments, [199]

Table 3.1: Test performance on 2D mazes of size 15×15 with **varying kernel sizes** F and best K setting for each F. Bold indicates best result across all F for each model and transition kernel. VIN performs worse with larger F, while GPPN is more robust when F is varied and actually works better with larger F.

		NEWS		Moore		Diff. Drive	
Model	F	%Opt	%Suc	%Opt	%Suc	%Opt	%Suc
VIN	3	93.4	93.5	90.5	91.3	**98.4**	**99.1**
VIN	5	**93.9**	**94.1**	**96.3**	**96.6**	96.4	98.6
VIN	7	92.7	93.0	95.1	95.6	92.2	96.2
VIN	9	86.8	87.8	92.0	93.0	91.2	95.2
VIN	11	87.6	88.3	92.7	93.8	87.9	93.8
GPPN	3	97.6	98.3	96.8	97.6	96.4	98.1
GPPN	5	98.6	99.0	98.4	99.1	98.7	99.5
GPPN	7	99.0	99.3	98.8	99.3	99.1	99.7
GPPN	9	99.0	99.4	**98.8**	**99.3**	**99.3**	**99.7**
GPPN	11	**99.2**	**99.5**	98.6	99.2	99.2	99.6

Table 3.2: Test performance on 2D mazes of size 15×15 with **varying kernel sizes** F and **iteration counts** K. "–" indicates the training diverged. GPPN outperforms VIN under best settings of (K, F), indicated in bold. By utilizing a larger F, GPPN can learn to more effectively propagate information spatially in a smaller number of iterations ($K \leq 10$) than VIN can.

		%Opt for NEWS					%Opt for Moore					%Opt for Differential Drive				
Model	K	$F=3$	$F=5$	$F=7$	$F=9$	$F=11$	$F=3$	$F=5$	$F=7$	$F=9$	$F=11$	$F=3$	$F=5$	$F=7$	$F=9$	$F=11$
VIN	5	55.6	87.7	84.6	86.3	86.6	75.0	86.7	88.9	92.0	92.3	74.8	91.9	91.5	91.2	87.9
VIN	10	79.0	83.3	92.2	86.8	86.7	90.5	91.4	95.1	89.4	92.7	92.4	96.1	92.2	84.0	64.4
VIN	15	91.3	92.9	92.7	85.4	87.6	88.7	89.6	92.4	90.0	91.0	96.7	96.4	90.1	65.2	23.0
VIN	20	93.4	**93.9**	91.4	86.3	85.5	80.9	92.8	90.7	89.1	90.4	97.7	94.8	89.0	40.0	22.3
VIN	30	71.2	92.8	84.5	86.5	86.4	80.5	**96.3**	92.5	91.7	89.1	**98.4**	95.9	89.5	–	–
GPPN	5	66.2	86.5	90.8	92.4	93.0	75.9	90.4	93.4	93.9	94.1	62.4	82.3	88.6	90.1	91.2
GPPN	10	91.2	96.1	97.1	97.6	97.7	93.3	96.5	97.4	97.6	97.4	87.7	95.4	96.1	97.0	97.4
GPPN	15	95.3	98.1	98.5	98.3	98.8	96.1	97.7	98.1	98.1	98.3	93.5	97.1	97.8	97.7	99.0
GPPN	20	97.4	98.4	99.0	99.0	**99.2**	96.8	98.4	98.5	98.7	98.6	95.8	97.9	98.4	98.4	98.9
GPPN	30	97.6	98.6	99.0	98.6	98.8	98.0	98.4	98.8	**98.8**	98.4	96.4	98.7	99.1	**99.3**	99.2

designed a multi-scale variant called the Hierarchical VIN. Hierarchical VINs rely on downsampling the maps into multi-scale hierarchies, and then doing VIN planning and up-scaling, progressively growing the map until it regains its original, un-downsampled size.

Another potential method to do long-range planning without requiring a multi-scale hierarchy is to instead increase the kernel size. An increased kernel size would cause the receptive field to grow more rapidly, potentially allowing the models to require fewer iterations K before reaching well-performing policies. In this section, we sought to test out the feasibility of increasing the kernel size of VINs and GPPNs. These results are summarized in Table 3.1. All the models were trained with the best K setting for each F and transition kernel. From the results, we can clearly see that GPPN can handle training with larger F values, and moreover, GPPN often performs better than VIN with larger values of F. In contrast, we can observe that VIN's performance drops significantly after its kernel size is increased more than 5, with its best performing settings being either 3 or 5 depending on the true transition model. These results show that GPPN can learn planning approximations that work with $F > 3$ much more stably than VIN, and could further suggest that GPPN can work as well as VIN with less iterations.

Varying Iteration Count K

Following the above results showing that GPPN benefits from increased F, we further evaluated the effect of varying both iteration count K and kernel size F on the VIN and GPPN models. Table 3.2 shows %Optimal and %Success results of VIN and GPPN on 15×15 2D mazes for different values of F and K. We can see from NEWS column in the table that GPPN with $F > 7$ can get results on par with the best VIN model with only $K = 5$ iterations. This shows that GPPN can learn to more effectively propagate information spatially in a smaller number of iterations than VIN can,

Table 3.3: Test performance on 2D mazes of size 15×15 with **varying iteration counts** K and best F setting for each K. Bold indicates best result across all K for each model and transition kernel. Generally, increasing K improves performance.

		NEWS		Moore		Diff. Drive	
Model	K	%Opt	%Suc	%Opt	%Suc	%Opt	%Suc
VIN	5	87.7	88.4	92.3	93.3	91.9	95.8
VIN	10	92.2	92.5	95.1	95.6	96.1	97.9
VIN	15	92.9	93.0	92.4	93.9	96.7	98.3
VIN	20	**93.9**	**94.1**	92.8	94.0	97.7	98.8
VIN	30	92.8	93.2	**96.3**	**96.6**	**98.4**	**99.1**
GPPN	5	93.0	94.3	94.1	96.1	91.2	95.6
GPPN	10	97.7	98.4	97.6	98.4	97.4	98.8
GPPN	15	98.8	99.2	98.3	98.9	99.0	99.6
GPPN	20	**99.2**	**99.5**	98.7	99.2	98.9	99.5
GPPN	30	99.0	99.3	**98.8**	**99.3**	**99.3**	**99.7**

Table 3.4: Test performance on 2D mazes of size 15×15 with **varying dataset sizes** N under best settings of (K, F) for each model. Both models improve with more training data (larger N). GPPN performs relatively better than VIN with less data, suggesting that the VIN architectural biases do not help generalization performance.

		NEWS		Moore		Diff. Drive	
N	Model	%Opt	%Suc	%Opt	%Suc	%Opt	%Suc
10k	VIN	90.3	90.6	88.1	90.5	97.5	98.4
10k	GPPN	**97.8**	**98.6**	**97.6**	**98.4**	**98.0**	**99.4**
25k	VIN	93.9	94.1	96.3	96.6	98.4	99.1
25k	GPPN	**99.2**	**99.5**	**98.8**	**99.3**	**99.3**	**99.7**
100k	VIN	97.3	97.3	97.1	97.5	98.9	99.4
100k	GPPN	**99.9**	**99.9**	**99.7**	**99.8**	**99.9**	**99.9**

and outperforms VIN even when VIN is given a much larger number of iterations. Additionally, we can see that VIN has significant trouble learning when both K and F are large in the differential drive mazes and to a lesser extent in the NEWS mazes.

Table 3.3 shows the results of VIN and GPPN with varying iteration counts K and the best F setting for each K. Owing to the larger kernel size, GPPN with smaller number of iterations $K \leq 10$ can get results on par with the best VIN model. Generally, both models benefit from a larger K (assuming the best F setting is used).

Different Maze Transition Kernels

From Tables 3.1 and 3.3, we can observe the performance of VIN and GPPN across a variety of different underlying groundtruth transition kernels (NEWS, Moore, and Differential Drive). From these results, we can see that GPPN consistently outperforms VIN on all the transition kernel types. An interesting observation is that VIN does very well at Differential Drive, consistently obtaining high results, although GPPN still does better than or on par with VIN. The reasons why VIN is so well suited to Differential Drive are not clear, and a preliminary analysis of VIN's feature weights and reward vectors did not reveal any intuition on why this is the case.

Effect of Dataset Size

A potential benefit of the stronger architectural biases of VIN might be that they can enable better generalization given less training data. In this section, we designed experiments that set out to test this hypothesis. We trained VINs and GPPNs on datasets with varying number of training samples for all three maze transition kernels, and the results are given in Table 3.4. We can see that GPPN consistently outperforms VIN across all dataset sizes and maze models. Interestingly, we can observe that the performance gap between VIN and GPPN is larger the less data there is, demonstrating the opposite effect to our hypothesis. This could suggest that the architectural biases do not in fact aid generalization performance, or that there is another problem, such as

Table 3.5: Mean and standard deviation %Opt after 30 epochs, taken over 3 runs, on 2D mazes of size 15×15. Bold indicates best result across all K for each model and transition kernel. The results were obtained using the best setting of F for each K and dataset size 100k. GPPN exhibits lower variance between runs.

		NEWS %Opt				Diff. Drive %Opt			
		Train		Val.		Train		Val.	
Model	K	mean	std	mean	std	mean	std	mean	std
VIN	5	90.1	0.1	90.1	1.5	88.4	1.0	95.4	1.1
VIN	10	92.8	0.6	92.7	1.4	92.3	0.5	93.9	0.2
VIN	15	93.4	1.2	94.2	0.6	**95.8**	**0.5**	**97.0**	**0.3**
VIN	20	**93.1**	**1.5**	**94.3**	**0.8**	96.4	0.2	96.8	1.1
GPPN	5	95.5	0.2	95.2	<0.1	93.8	0.2	93.4	<0.1
GPPN	10	99.1	0.1	99.0	<0.1	98.7	0.1	98.2	0.2
GPPN	15	99.6	<0.1	99.6	<0.1	99.4	0.1	99.3	0.1
GPPN	20	**99.7**	**<0.1**	**99.7**	**0.1**	**99.8**	**<0.1**	**99.7**	**0.1**

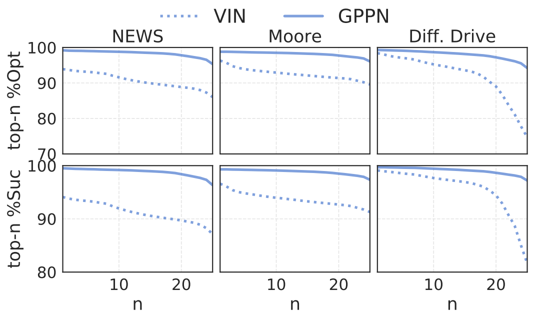


Figure 3.2: The y-axis is the average Test %Opt (or %Suc) of the top-n hyperparameter settings (K, F) over $K \in \{5, 10, 15, 20, 30\}$ and $F \in \{3, 5, 7, 9, 11\}$. The results are on 2D mazes of size 15×15. These plots measure how stable the performance of each model is to hyperparameter changes as we increase the number of hyperparameter settings considered. GPPN exhibits less hyperparameter sentivitiy.

perhaps the difficulty of optimizing VIN, that overshadows the benefit that the inductive bias could potentially provide.

Random Seed and Hyperparameter Sensitivity

The hypothesis this section sought to verify was whether the particular recurrent-convolutional form of the VIN did indeed negatively affect its optimization, as many ungated recurrent updates suffer from optimization problems including training instability and higher sensitivity to weight initialization and hyperparameters due to gradient scaling problems [156].

We test each architecture's sensitivity to random seeds by running several experiments with the same hyperparameters but different random seeds, and measuring the variance in their final performance. These results are reported in Table 3.5. The results show that GPPN gets consistently lower variance than VIN over different random seed initializations, supporting the hypothesis that the LSTM update enables more training stability and easier optimization than the ungated recurrent update in VIN.

We additionally test hyperparameter sensitivity in Figure 3.2. We take all the results obtained on a hyperparameter sweep over settings (K, F) where K was varied over $K \in \{5, 10, 15, 20, 30\}$ and F was varied over $F \in \{3, 5, 7, 9, 11\}$. We then rank these results, and the x-axis is the top-x ranked hyperparameter settings and the corresponding y-axis is the average %Opt/%Suc of those x settings. This plot thus measures how stable the performance of the architecture is to hyperparameter changes as the number of hyperparameter settings we consider grows. Therefore, architectures whose average top-x ranked performance remains high and relatively flat demonstrates that good performance with the architecture can be obtained with many different hyperparameter settings. This suggests that these models are both easier to optimize and consistently better than alternatives, and higher performance was not due to a single lucky hyperparameter setting. We can

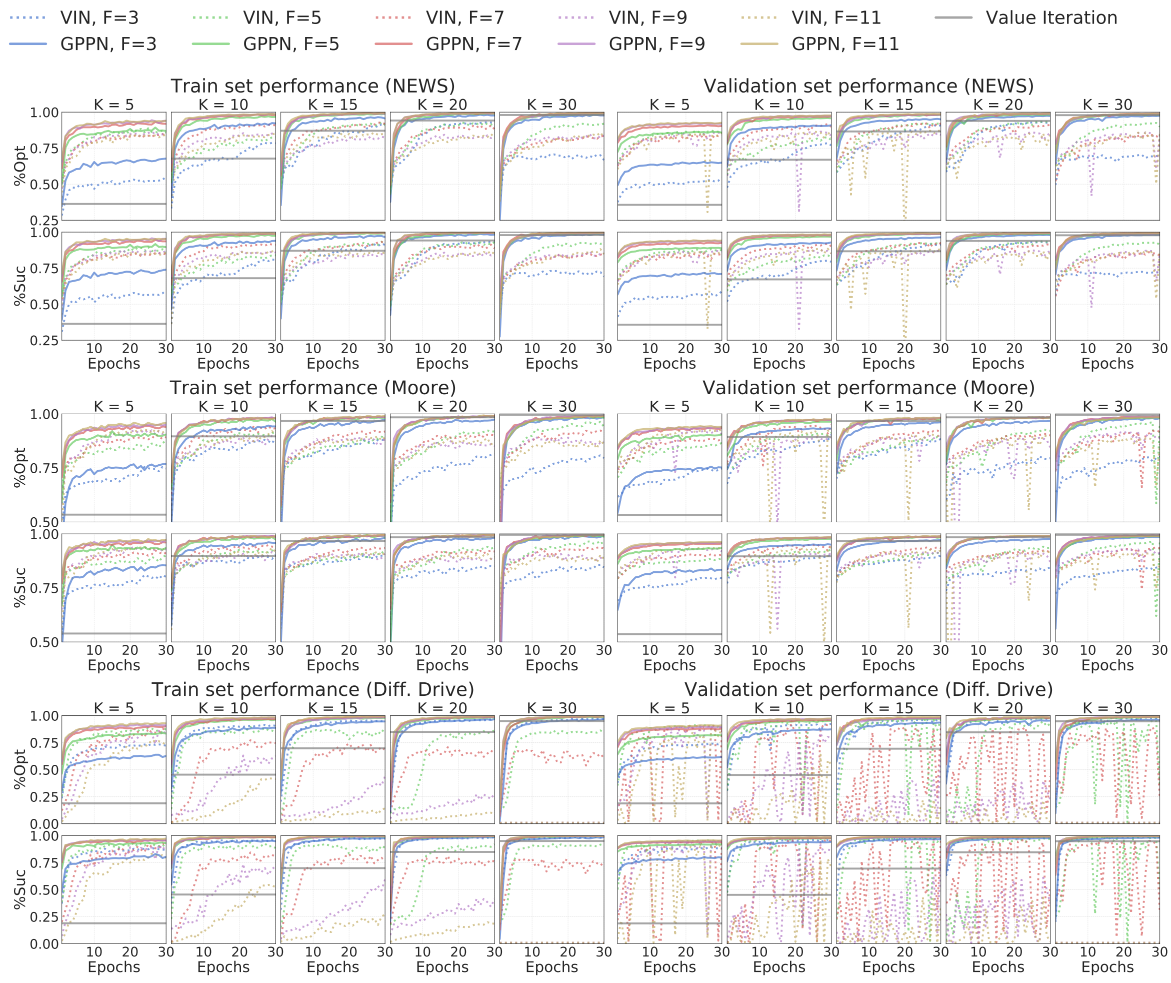


Figure 3.3: Performance on 2D mazes of size 15 × 15 with varying iteration counts K and kernel sizes F . All models are trained using dataset size 25k. VIN exhibits higher training instability, its performance often oscillating between epochs.

see from the figures that the performance of GPPN is clearly both higher and more stable over hyperparameter settings than VIN.

In Figure 3.3, we plot the learning curves for VIN and GPPN on 2D mazes with varying K and F. These plots show that VIN's performance often oscillates between epochs (especially for larger kernel sizes $F > 3$), while GPPN is much more stable. Learning curves for other experiments showing a similar result are included in the Appendix. The training stability of GPPN provides more evidence to the hypothesis that GPPNs are simpler to optimize than VINs and consistently outperform them.

Learning Speed

In this section, we examine whether VINs or GPPNs learn faster. To do this, we measure the number of training epochs (passes over the entire dataset) that it takes for each model to reach a specific %Opt for the first time. These results are reported in Table 3.6. We can see from this table that GPPN learns significantly faster, often reaching 95% within 5-6 epochs. Comparatively, VIN sometimes never reaches 95%, as is the case for the NEWS mazes, or it takes 2-5 times as many epochs. This is the case even on the Differential Drive mazes, where VIN takes 2-3 times longer to train despite also getting high final performance.

Larger Maze Size

To test whether the improved performance GPPN has persists even on larger, more challenging mazes, we evaluated the models on a dataset of mazes of size 28 × 28, and varied $K \in \{14, 28, 56\}$

Table 3.6: The number of epochs it takes for each model to attain a certain %Opt (50%, 75%, 90%, 95%) on the validation set under best settings of (K, F). The results are on 2D mazes of size 15×15. GPPN learns faster.

	NEWS				Moore				Diff. Drive			
Model	50	75	90	95	50	75	90	95	50	75	90	95
VIN	1	6	17	–	1	1	11	23	2	3	5	14
GPPN	1	1	3	5	1	1	3	5	1	2	3	6

Table 3.7: Test performance on 2D mazes with **varying maze sizes** $m \times m$ under best settings of (K, F) for each model. For the larger 28×28 maze, we train for 100 epochs and sweep over $K \in \{14, 28, 56\}$ to account for longer trajectories required to solve some mazes. GPPN performs better.

		NEWS		Moore		Diff. Drive	
m	Model	%Opt	%Suc	%Opt	%Suc	%Opt	%Suc
15	VIN	93.9	94.1	96.3	96.6	98.4	99.1
15	GPPN	**99.2**	**99.5**	**98.8**	**99.3**	**99.3**	**99.7**
28	VIN	93.0	93.2	95.0	95.8	93.8	96.8
28	GPPN	**98.3**	**98.9**	**97.8**	**98.7**	**99.0**	**99.6**

(Table 3.7). We used a training dataset size of 25k. GPPN outperformed VIN by a significant margin (3-6% for %Opt and %Suc) for all cases except Diff. Drive 15×15, where the gap was closer (GPPN 99.3 vs. VIN 98.3 for %Opt).

3D ViZDoom Experiments

In the 3D ViZDoom experiments, the state vector consists of RGB images showing the first-person view of the environment at each position and orientation, instead of the top-down 2D maze design (represented by a binary $m \times m$ matrix) as in the 2D maze experiments. To process the map images, we use a Convolutional Neural Network [117] consisting of two convolutional layers: first layer with 32 filters of size 8×8 and a stride of 4, and second layer with 64 filters of size 4×4 with a stride 2×2, followed by a linear layer of size 256.[2] The 256-dimensional representation for all the 4 orientations at each location is concatenated to create a 1024-dimensional representation. These representations of each location are then stacked at the corresponding x-y coordinate to create a map representation of size $1024 \times m \times m$. The map representation is then passed through two more convolutional layers (first layer with 64 filters and the second layer with 1 filter, both of size 3×3 and a stride of 1) to predict a maze design matrix of size $1 \times m \times m$, which is trained using an auxillary binary cross-entropy loss. The predicted maze design is then stacked with the goal map and passed to the VIN or GPPN module in the same way as the 2D experiments.

The 3D ViZDoom results are summarized in Table 3.8. **%Acc** is the accuracy for predicting the top-down 2D maze design from first-person RGB images. Learning to plan in the 3D environments is more challenging due to the difficulty of simultaneously optimizing both the original planner loss and the auxiliary maze prediction loss. We can see that when %Acc is low, i.e., the planner module must rely on a noisy maze design, then the planner metrics %Opt and %Suc also suffer. We observe that VIN is more prone to overfitting on the training dataset: its validation %Acc is low ($< 91\%$) for all three transition kernels, whereas GPPN achieves higher validation %Acc on NEWS and Moore. However, GPPN also overfits on the Differential Drive.

3.4 Active Neural Localizer

Bayesian filters [65] are used to probabilistically estimate a dynamic system's state using observations from the environment and actions taken by the agent. Let y_t be the random variable representing the state at time t. Let s_t be the observation received by the agent and a_t be the action taken by the agent at time step t. At any point in time, the probability distribution over y_t conditioned

[2]This architecture was adapted from a previous work which is shown to perform well at playing deathmatches in Doom [114].

Table 3.8: Performance on **3D ViZDoom mazes**. %Acc is accuracy for predicting the top-down 2D maze design from first-person RGB maze images. When %Acc is low, then the model must use a noisy maze design from which to plan, so %Opt and %Suc suffer as well. The results were obtained using $K = 30$, the best setting of F for each transition kernel, a smaller dataset size 10k (due to memory and time constraints), a smaller learning rate 5e-4, and 100 training epochs. VIN is more prone to overfitting: its validation %Acc is low for all three transition kernels, while GPPN achieves higher validation %Acc on NEWS and Moore.

		Train			Val			Test	
Kernel	Model	%Acc	%Opt	%Suc	%Acc	%Opt	%Suc	%Opt	%Suc
NEWS	VIN	99.9	82.3	83.0	81.5	80.8	81.5	79.0	79.7
NEWS	GPPN	99.9	99.4	99.7	94.9	93.2	94.9	94.1	95.9
Moore	VIN	99.6	86.5	88.9	89.1	86.7	89.1	84.6	87.6
Moore	GPPN	99.6	98.1	99.4	97.4	95.3	97.4	94.5	97.2
Diff. Drive	VIN	100.0	99.4	99.7	90.5	89.0	90.5	96.9	97.9
Diff. Drive	GPPN	99.8	99.5	100.0	85.0	81.0	85.0	91.4	96.0

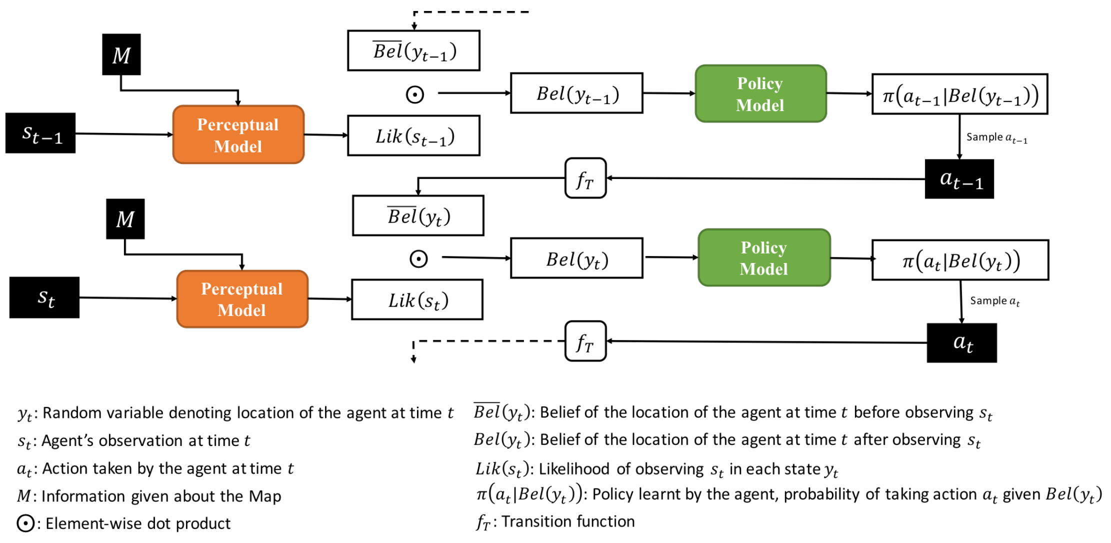

M
s_{t-1}
Perceptual Model
$\overline{Bel}(y_{t-1})$
$\odot$
$Lik(s_{t-1})$
$Bel(y_{t-1})$
Policy Model
$\pi(a_{t-1}|Bel(y_{t-1}))$
Sample a_{t-1}
a_{t-1}
f_T
M
s_t
Perceptual Model
$\overline{Bel}(y_t)$
$\odot$
$Lik(s_t)$
$Bel(y_t)$
Policy Model
$\pi(a_t|Bel(y_t))$
Sample a_t
a_t
f_T
y_t: Random variable denoting location of the agent at time t
s_t: Agent's observation at time t
a_t: Action taken by the agent at time t
M: Information given about the Map
$\odot$: Element-wise dot product
$\overline{Bel}(y_t)$: Belief of the location of the agent at time t before observing s_t
$Bel(y_t)$: Belief of the location of the agent at time t after observing s_t
$Lik(s_t)$: Likelihood of observing s_t in each state y_t
$\pi(a_t|Bel(y_t))$: Policy learnt by the agent, probability of taking action a_t given $Bel(y_t)$
f_T: Transition function

Figure 3.4: The architecture of the proposed model. The perceptual model computes the likelihood of the current observation in all possible locations. The belief of agent's location is propagated through time by taking element-wise dot-product with the likelihood. The policy model learns a policy to localize accurately while minimizing the number of steps required for localization. See text for more details.

over past observations $s_{1:t-1}$ and actions $a_{1:t-1}$ is called the *belief*, $Bel(y_t) = p(y_t|s_{1:t-1}, a_{1:t-1})$ The goal of Bayesian filtering is to estimate the belief sequentially. For the task of localization, y_t represents the location of the agent, although in general it can represent the state of any object(s) in the environment. Under the Markov assumption, the belief can be recursively computed using the following equations:

$$\overline{Bel}(y_t) = \sum_{y_{t-1}} p(y_t|y_{t-1}, a_{t-1})Bel(y_{t-1}), \quad Bel(y_t) = \frac{1}{Z}Lik(s_t)\overline{Bel}(y_t),$$

where $Lik(s_t) = p(s_t|y_t)$ is the likelihood of observing s_t given the location of the agent is y_t, and $Z = \Sigma_{y_t} Lik(s_t)\overline{Bel}(y_t)$ is the normalization constant. The likelihood of the observation, $Lik(s_t)$ is given by the *perceptual model* and $p(y_t|y_{t-1}, a_{t-1})$, i.e. the probability of landing in a state y_t from y_{t-1} based on the action, a_{t-1}, taken by the agent is specified by a state transition function, f_t. The belief at time $t = 0$, $Bel(y_0)$, also known as the *prior*, can be specified based on prior knowledge about the location of the agent. For global localization, prior belief is typically uniform over all possible locations of the agent as the agent position is completely unknown.

We now first define the Active Neural Localizer architecture and then present experiments validating its performance against previous baselines. The Active Neural Localizer utilizes a belief state represented as an $O \times M \times N$ tensor, where $(i, j, k)^{th}$ element denotes the belief of the agent being in the corresponding state, $Bel(y_t = i, j, k)$. This kind of grid-based representation of belief is popular among localization methods as it offers several advantages over topological representations [22, 65]. Let $Lik(s_t) = p(s_t|y_t)$ be the likelihood of observing s_t given the location of the agent is y_t. The likelihood of an observation in a certain state is also represented by an $O \times M \times N$ tensor, where $(i, j, k)^{th}$ element denotes the likelihood of the current observation, s_t, given that the agent's state is $y_t = i, j, k$. We refer to these tensors as Belief Map and Likelihood Map in the rest of the chapter.

3.4.1 Model Architecture

The overall architecture of the Active Neural Localizer is shown in Figure 3.4. It has two main components: the perceptual model and the policy model. At each timestep t, the *perceptual model* takes in the agent's observation, s_t and outputs the Likelihood Map $Lik(s_t)$. The belief is propagated through time by taking an element-wise dot product with the Likelihood Map at each timestep. Let $\overline{Bel}(y_t)$ be the Belief Map at time t before observing s_t. Then the belief, after observing s_t, denoted by $Bel(y_t)$, is calculated as follows:

$$Bel(y_t) = \frac{1}{Z}\overline{Bel}(y_t) \odot Lik(s_t)$$

where $\odot$ denotes the Hadamard product, $Z = \sum_{y_t} Lik(s_t)\overline{Bel}(y_t)$ is the normalization constant.

The Belief Map, after observing s_t, is passed through the *policy model* to obtain the probability of taking any action, $\pi(a_t|Bel(y_t))$. The agent takes an action a_t sampled from this policy. The Belief Map at time $t+1$ is calculated using the transition function (f_T), which updates the belief at each location according to the action taken by the agent, i.e. $p(y_{t+1}|y_t, a_t)$. The transition function here is the "Differential Drive" kernel described in Section 3.2.2. For 'turn left' and 'turn right' actions, the transition model swaps the belief in each orientation, where for the 'move forward' action, the belief values are shifted one unit according to the orientation. If the next unit is an obstacle, then the value doesn't shift, indicating a collison.

3.4.2 Model Components

Perceptual Model The perceptual model computes the feature representation from the agent's observation and the states given in the map information. The likelihood of each state in the map information is calculated by taking the cosine similarity of the feature representation of the agent's observation with the feature representation of the state. Cosine similarity is commonly used for computing the similarity of representations [141, 91] and has also been used in this context in localization [24]. The benefits of using cosine similarity over the dot-product operation have been highlighted by [32].

In the 2D environments, the observation is used to compute a one-hot vector of the same dimension representing the depth which is used as the feature representation directly. This resultant Likelihood map has uniform non-zero probabilities for all locations having the observed depth and zero probabilities everywhere else. For the 3D environments, the feature representation of each observation is obtained using a trainable deep convolutional network [116]. Figure 3.1 shows examples of the agent observation and the corresponding Likelihood Map computed in both 2D and 3D environments.

Policy Model The policy model gives the probablity of the next action given the current belief of the agent. It is trained using reinforcement learning, specifically Asynchronous Advantage Actor-Critic (A3C) [136] algorithm. The belief map is stacked with the map design matrix and passed through 2 convolutional layers followed by a fully-connected layer to predict the policy as well as the value function. The policy and value losses are computed using the rewards observed by the agent and backpropagated through the entire model.

3.4.3 Baselines

Markov Localization [63] is a passive probabilistic approach based on Bayesian filtering. We use a geometric variant of Markov localization where the state space is represented by a fine-grained, regularly spaced grid, called position probability grids [22], similar to the state space in the proposed model. Grid-based state space representations are known to offer several advantages over topological

Table 3.9: Results on the 2D Environments. 'Time' refers to the number of seconds required to evaluate 1000 episodes with the corresponding method and 'Acc' stands for accuracy over 1000 episodes.

Env		2D Mazes						All
Map Size		7x7		15x15		21x21		
Episode Length		15	30	20	40	30	60	
Markov	Time	12	15	29	31	49	51	31.2
Localization	Acc	0.334	0.529	0.351	0.606	0.414	0.661	0.483
Active Markov	Time	29	53	72	165	159	303	130.2
Localization (Fast)	Acc	0.436	0.619	0.468	0.657	0.512	0.735	0.571
Active Markov	Time	1698	3066	3791	8649	8409	13554	6527.8
Localization (Slow)	Acc	0.854	0.938	0.846	**0.984**	0.845	0.958	0.904
Active Neural	Time	22	34	44	66	82	124	62.0
Localization	Acc	**0.936**	**0.939**	**0.905**	0.939	**0.899**	**0.984**	**0.934**

representations [22, 65]. In the passive localization approaches, actions taken by the agent are random.

Active Markov Localization (AML) [64] is the active variant of Markov Localization where the actions taken by the agent are chosen to maximize the ratio of the 'utility' of the action to the 'cost' of the action. The 'utility' of an action a at time t is defined as the expected reduction in the uncertainity of the agent state after taking the action a at time t and making the next observation at time $t+1$: $U_t(a) = H(y_t) - \mathbb{E}_a[H(y_{t+1})]$, where $H(y)$ denotes the entropy of the belief: $H(y) = -\sum_y Bel(y) \log Bel(y)$, and $\mathbb{E}_a[H(y_{t+1})]$ denotes the expected entropy of the agent after taking the action a and observing y_{t+1}. The 'cost' of an action refers to the time needed to perform the action. In our environment, each action takes a single time step, thus the cost is constant.

We define a generalized version of the AML algorithm. The utility can be maximized over a sequence of actions rather than just a single action. Let $a^* \in \mathbb{A}^{n_l}$ be the action sequence of length n_l that maximizes the utility at time t, $a^* = \arg\max_a U_t(a)$ (where $\mathbb{A}$ denotes the action space). After computing a^*, the agent need not take all the actions in a^* before maximizing the utility again. This is because new observations made while taking the actions in a^* might affect the utility of remaining actions. Let $n_g \in \{1, 2, ..., n_l\}$ be the number of actions taken by the agent, denoting the greediness of the algorithm. Due to the high computational complexity of calculating utility, the agent performs random actions until the belief is concentrated on n_m states (ignoring beliefs under a certain threshold). The complexity of the generalized AML is $\mathcal{O}(n_m(n_l - n_g)|\mathbb{A}|^{n_l})$. Given sufficient computational power, the optimal sequence of actions can be calculated with n_l equal to the length of the episode, $n_g = 1$, and n_m equal to the size of the state space.

In the original AML algorithm, the utility was maximized over single actions, i.e. $n_l = 1$ which also makes $n_g = 1$. The value of n_m used in their experiments is not reported, however they show an example with $n_m = 6$. We run AML with all possible combination of values of $n_l \in \{1, 5, 10, 15\}$, $n_g \in \{1, n_l\}$ and $n_m = \{5, 10\}$ and define two versions: (1) Active Markov Localization (Fast): Generalized AML algorithm using the values of n_l, n_g, n_m that maximize the performance while keeping the runtime comparable to Active Neural Localization, and (2) Active Markov Localization (Slow): Generalized AML algorithm using the values of n_l, n_g, n_m which maximize the performance while keeping the runtime for 1000 episodes below 24hrs (which is the training time of the proposed model) in each environment.

The perceptual model for both Markov Localization and Active Markov Localization needs to be specified separately. For the 2D environments, the perceptual model uses 1-hot vector representation of depth. For the 3D Environments, the perceptual model uses a pretrained Resnet-18 [81] model to calculate the feature representations for the agent observations and the memory images.

3.4.4 Results

2D Environments For the 2D maze domain, we run all models on mazes having size 7×7, 15×15 and 21×21 with varying episode lengths. We train all the models on random Kruskal-generated mazes and test on a fixed set of 1000 mazes (different from the mazes used in training). The results on the 2D mazes are shown in Table 3.9. As seen in the table, the proposed model, Active Neural

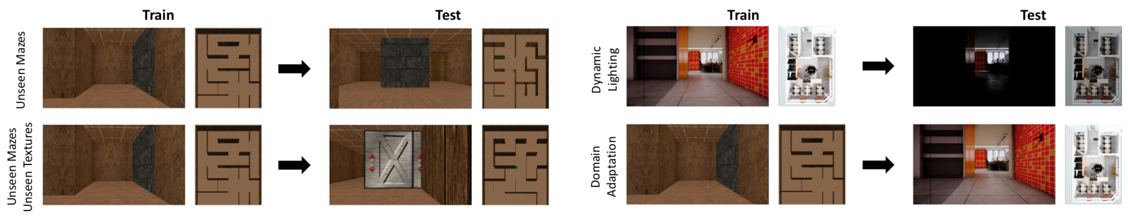


Figure 3.5: Different experiments in the 3D Environments. Refer to the text for more details.

Table 3.10: Results on the 3D environments. 'Time' refers to the number of seconds required to evaluate 1000 episodes with the corresponding method and 'Acc' stands for accuracy over 1000 episodes.

Env		ViZDoom Mazes						Unreal3D with lights		All	Domain adaptation
Evaluation Setting		Unseen Mazes Seen Textures			Unseen mazes Unseen textures			With lights	Without lights		ViZDoom to Unreal3D
No. of landmarks		10	5	0	10	5	0				
Markov Localization (Resnet)	Time	2415	2470	2358	2580	2509	2489	2513	2541	2484.4	-
	Acc	0.716	0.657	0.641	0.702	0.669	0.652	0.517	0.249	0.600	-
Active Markov Localization (Fast)	Time	14231	12409	11662	15738	12098	11761	11878	5511	11911.0	-
	Acc	0.741	0.701	0.669	0.745	0.687	0.689	0.546	0.279	0.632	-
Active Markov Localization (Slow)	Time	48291	47424	43096	48910	44500	44234	47962	11205	41952.8	-
	Acc	0.759	0.749	0.694	0.787	0.730	0.720	0.577	0.302	0.665	-
Active Neural Localization	Time	297	300	300	300	300	301	2750	2699	905.9	2756
	Acc	**0.889**	**0.859**	**0.852**	**0.858**	**0.839**	**0.871**	**0.934**	**0.505**	**0.826**	**0.921**

Localization, outperforms all the baselines on average. The proposed method achieves a higher overall accuracy than AML (Slow) while being 100 times faster. Note that the runtime of AML for 1000 episodes is comparable to the total training time of the proposed model. The long runtime of AML (Slow) makes it infeasible to be used in real-time in many cases. When AML has comparable runtime, its performance drops by about 37% (AML (Fast)). We also observe that the difference in the performance of Active Neural Localization and baselines is higher for smaller episode lengths. This indicates that Active Neural Localization is more efficient (meaning it requires fewer actions to localize) in addition to being more accurate.

3D Environments All the mazes in the 3D ViZDoom domain are of size 70×70 while the office environment environment is of size 70×50. The agent location is a continuous value in this range. Each cell roughly corresponds to an area of 40cm×40cm in the real world. The set of memory images correspond to only about 6% of the total states. Likelihood of rest of the states are obtained by bilinear smoothing. All episodes have a fixed length of 30 actions. Although the size of the Office Map is 70×50, we represent Likelihood and Belief by a 70×70 in order to transfer the model between both the 3D environments for domain adaptation. We also add a Gaussian noise of 5% standard deviation to all translations in 3D environments.

In the **ViZDoom** domain, we vary the difficulty of the environment by varying the number of *landmarks* in the environment. Landmarks are defined to be walls with a unique texture. Each landmark is present only on a single wall in a single cell in the maze grid. All the other walls have a common texture making the map very ambiguous. We expect landmarks to make localization easier as the likelihood maps should have a lower entropy when the agent visits a landmark, which consequently should reduce the entropy of the Belief Map. We run experiments with 10, 5 and 0 landmarks. The textures of the landmarks are randomized during training. This technique of domain randomization has shown to be effective in generalizing to unknown maps within the simulation environment [113] and transferring from simulation to real-world [204]. In each experiment, the agent is trained on a set of 40 mazes and evaluated in two settings: (1) Unseen mazes with seen textures: the textures of each wall in the test set mazes have been seen in the training set, however the map design of the test set mazes are unseen and (2) Unseen mazes with unseen textures: both the textures and the map design are unseen. We test on a set of 20 mazes for each evaluation setting. Figure 3.5 shows examples for both the settings.

In the **Unreal3D** environment, we test the effectiveness of the model in adapting to dynamic lightning changes. We modified the the Office environment using the Unreal Game Engine Editor to create two scenarios: (1) Lights: where all the office lights are switched on; (2) NoLights: where all the office lights are switched off. Figure 3.5 shows sample agent observations with and without lights at the same locations. To test the model's ability to adapt to dynamic lighting changes, we train the model on the Office map with lights and test it on same map without lights. The memory images provided to the agent are taken while lights are switched on. Note that this is different as compared to the experiments on unseen mazes in ViZDoom environments, where the agent is given memory images of the unseen environments.

The results for the 3D environments are shown in Table 3.10 and an example of the policy execution is shown in Figure 3.6[3]. The proposed model significantly outperforms all baseline models in all evaluation settings with the lowest runtime. We see similar trends of runtime and accuracy trade-off between the two versions of AML as seen in the 2D results. The absolute performance of AML (Slow) is rather poor in the 3D environments as compared to 2D mazes. This is likely due to the decrease in value of look-ahead parameter, n_l, to 3 and the increase in value of the greediness hyper-parameter, n_g to 3, as compared to $n_l = 5, n_g = 1$ in Maze 2D, in order to ensure runtimes under 24hrs.

The Active Neural Localization model performs better on the realistic Unreal environment as compared to the ViZDoom environment, as most scenes in the Unreal environment consists of unique landmarks while ViZDoom environments consists of random mazes with the same texture except those of the landmarks. In the ViZDoom domain, the model is able to generalize well to not only unseen map design but also to unseen textures. However, the model doesn't generalize well to dynamic lighting changes in the Unreal3D environment. This highlights a current limitation of RGB image-based localization approaches as compared to depth-based approaches, as depth sensors are invariant to lighting changes.

Domain Adaptation We also test the ability of the proposed model to adapt between different simulation environments. The model trained on the ViZDoom domain is directly tested on the Unreal3D Office Map without any fine-tuning. The results in Table 3.10 show that the model is able to generalize well to the Unreal environment from the Doom environment. We believe that the policy model generalizes well because (1) the representation of belief and map design is learnt jointly in many randomized environments and (2) the policy model is based only on the belief and the map design. Similarly the perceptual model generalizes well because it was trained on environments with random textures.

3.5 Discussion

To conclude, in this chapter we proposed two differentiable submodules capable of enhancing the Neural Map inter-episodic memory described in the previous chapter with two core capabilities: the first is the Gated Path Planning Network, a planning module capable of doing efficient model-based planning and search, and the second is Active Neural Localization, a state inference module (equivalently, localizer in the setting of embodied environment) that not only disambiguates agent state quickly but chooses actions to actively reduce uncertainty. The GPPN motivated itself by re-formulating VIN as a convolutional-recurrent network, replacing the unconventional recurrent value-iteration-like update with a well-established gated LSTM recurrent operator. The Active Neural Localizer uses structured components for Bayes filter-like belief propagation and learns a policy based on the belief to localize accurately and efficiently. This allows the policy and observation models to be trained jointly using reinforcement learning. We showed the effectiveness of the proposed model on a variety of challenging 2D and 3D environments including a realistic map in the Unreal environment. Both of these submodules enabled learning initially unknown MDP parameters directly from data, allowing the learning of model-based planning directly from data, without needing a large amount of hand-crafted domain-specific knowledge typically provided by an agent designer. We presented experimental results comparing GPPN and Active Neural Localization on challenging 2D and 3D maze tasks, isolating the evaluation of each component so that they can be ablated against previous baselines directly. The results demonstrate that (1) the GPPN learns faster, generalizes better with less data, and achieves improved and more consistent results when compared to VIN and (2) that Active Neural Localization consistently

[3]Policy execution videos can be seen at `https://tinyurl.com/y8gp3mkh`

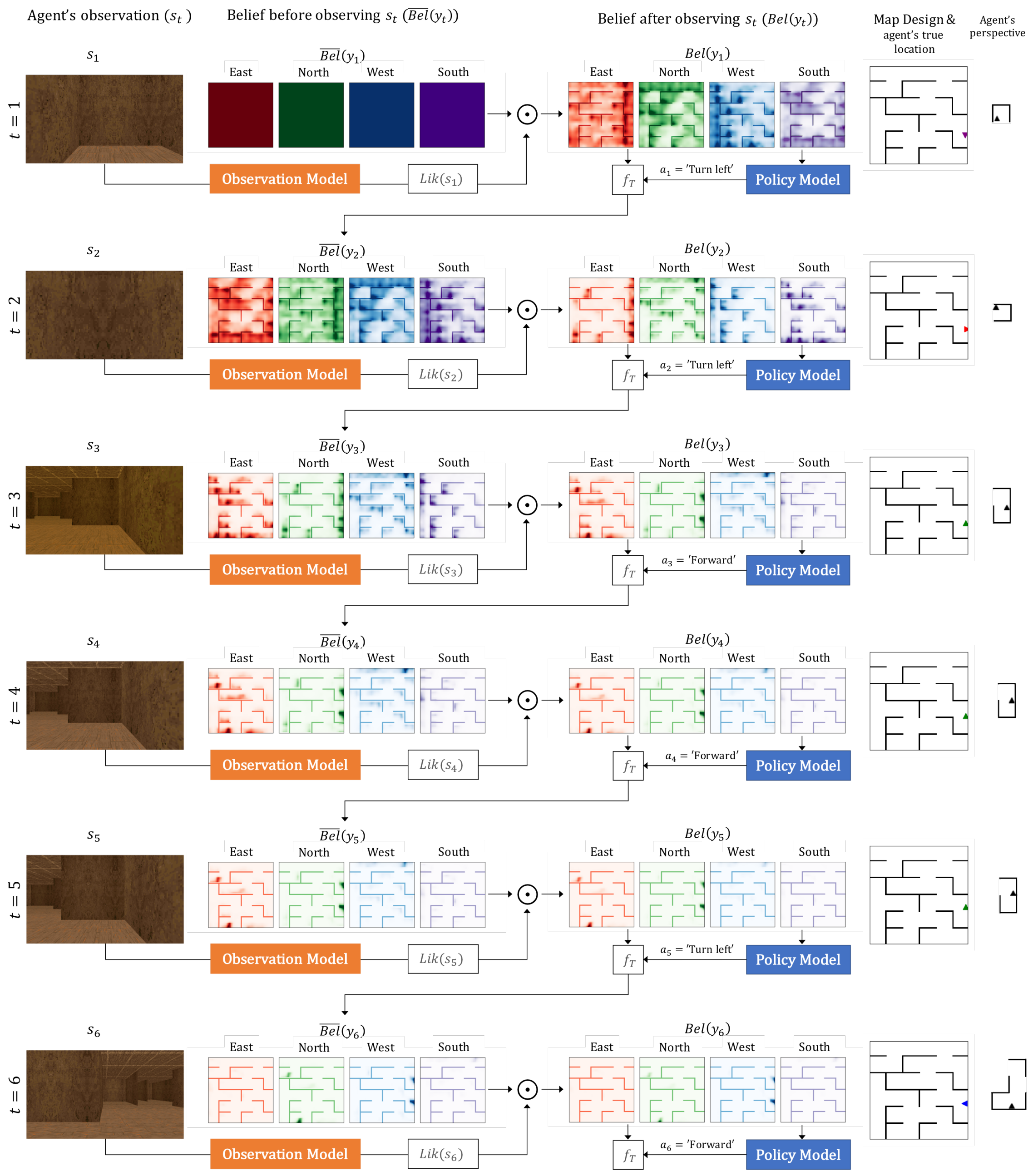


Figure 3.6: An example of the policy execution and belief propagation in the ViZDoom domain. The rows shows consecutive time steps in a episode. The columns show Agent's observation, the belief of its location before and after making the observation, the map design and agent's perspective of the world. Agent's true location is also marked in the map design (not visible to the agent). Belief maps show the probability of being at a particular location. Darker shades imply higher probability. The belief of its orientation and agent's true orientation are also highlighted by colors. For example, the Red belief map shows the probability of agent facing east direction at each x-y coordinate. Note that map design is not a part of the Belief Maps, it is overlayed on the Belief Maps for better visualization. At all time steps, all locations which look similar to agent's perspective have high probabilities in the belief map. The example shows the importance deciding actions while localizing. At $t = 3$, the agent is uncertain about its location as there are 4 positions with identical perspectives. The agent executes the optimal set of action to reduce its uncertainity, i.e. move forward and turn left, and successfully localizes.

outperforms the baseline models while being order of magnitudes faster, and is also capable of proficient zero-shot transfer to new domains when trained using random textures. In conclusion, we have successfully designed planning and localization submodules siginificantly outperforming the previous state-of-the-art while capable of being integrated directly with the inter-episodic Neural Map. This works towards our goal of a monolithic agent architecture for meta-learning embodied environments, capable of memorization, planning and state inference all in an end-to-end manner, requiring a minimal amount of domain-knowledge for the agent designer.

We now work towards addressing the outstanding missing capability of our potential monolithic architecture: the ability for the agent to localize successfully without a map. This ability is distinct from the localization described in this chapter, which relied on pre-specified map information. In this new setting, termed Simultaneous Localization and Mapping (SLAM), the agent both needs to answer how to perform its own localization as well as how the observations it has seen can be composed into a map. While algorithms to accomplish the seemingly competing SLAM objectives have been developed extensively [52][140][139], a completely end-to-end differentiable architecture was notably absent. Therefore in the next chapter, inspired by traditional graph-optimization methods from the literature [111], we design a differentiable architecture to perform SLAM, the Neural Graph Optimizer, and demonstrate that it outperforms traditional and deep-learning-based approaches to localization on embodied environments.

Chapter 4

Towards End-to-End Simultaneous Localization and Mapping

In the previous two chapters we first described an inter-episodic memory designed specifically for embodied applications and then designed submodules that achieve state-of-the-art differentiable planning and state inference capabilities. The defined memory, planning and state inference architectural components could potentially be combined into a full, monolithic agent architecture specialized for few-shot meta-learning in any embodied environment, but such an architecture would currently lack one outstanding core component: localization without a map, a critical prerequisite to the original Neural Map inter-episodic memory presented in the first chapter which had relied on an oracle to provide location or ego-motion estimates. While we can use any extant off-the-shell localization method, these are often optimization-based [111] and are not in general easily differentiable, if at all. Having a completely differentiable agent architecture is appealing as it enables the least amount of domain knowledge required, meaning the distributed representations can (1) be learnt directly from task and (2) learn cohesive representations such that they are adapted to the computational processes that are performed on them, e.g. observations features can potentially be learnt in such in a way that the GPPN requires fewer iterations to compute a plan. Therefore in this chapter, we move towards a completely differentiable submodule capable of doing localization without any map provided.

The stand-alone ability for an agent to localize itself within an environment is a crucial prerequisite for many real-world applications, such as household robots [202], autonomous drones [61], and augmented and virtual reality applications. In most cases, the main challenge for an agent localizing itself is that: the agent is not provided with a map of the environment and therefore the agent must simultaneously map the environment and localize itself within the incomplete map it has produced. Owing to its large practical importance, a wide variety of algorithms to solve Simultaneous Localization and Mapping (SLAM) task have been developed over a long history [202, 23], with modern methods achieving impressive accuracy and real-time performance [138, 111, 140, 52]. These methods still have several shortcomings, owing mainly to the hand-engineered features, dense matching, and heuristics used in the design of these algorithms. For example, most methods are brittle in certain scenarios, such as varying lighting conditions (e.g. changing time of day), different weather conditions or seasons [179], repetitive structures, textureless objects, extremely large viewpoint changes, dynamic elements within the environment, and faulty sensor calibration [23]. Because these situations are common in real-world scenarios, robust applications of those systems are difficult. The brittleness and reduced generalization of current methods can be viewed as a potential consequence of the non-differentiability of these methods, as the features must be hand-designed and tested by domain experts.

In this chapter we work towards a fully differentiable SLAM, developing a method which can be made more robust to the common situations where previous SLAM algorithms typically degrade. To do this, we formulate a novel neural network architecture called "Neural Graph Optimizer". The Neural Graph Optimizer consists of differentiable analogues of the common types of subsystems used in modern SLAM algorithms, such as a local pose estimation model, a pose selection module (key frame selection, essential graph), and a graph optimization process. Because each component in the system is differentiable, the entire architecture can be trained in an end-to-end fashion, enabling the network to learn invariances to the types of scenarios observed during training, as

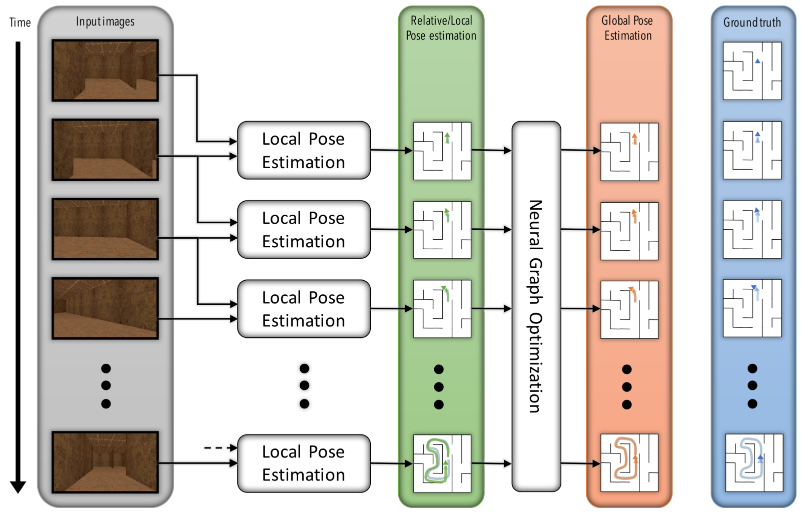


Figure 4.1: Components of the proposed model along with sample input, output and ground truth. The Local Pose Estimation model predicts the relative pose change between consecutive observations and Neural Graph Optimization model jointly optimizes the predictions of the Local Pose Estimation model to predict global pose changes. The local pose estimates, global pose estimates, and ground truth trajectory are shown in green, orange and blue, respectively.

well as enabling seamless composition with the rest of the submodules defined in previous chapters. To demonstrate the ability of our method to learn pose estimation, we use trajectories sampled from several simulated environments. The first environment is a 2D maze where the agent has a single-pixel row-scan as input. We then scale the model up to 3D mazes based on the ViZDoom environment [105], where the agent receives an image of the first-person view of the world as input.

4.1 Background & Related Work

SLAM is a process in which an agent needs to localize itself in an unknown environment and build a map of this environment at the same time, with uncertainties in both its motions and observations. SLAM has evolved from filter-based to graph-based (optimization-based) approaches. Some EKF-based systems have demonstrated state-of-the-art performance, such as the Multi-State Constraint Kalman Filter [138], the VIN [108], and the system of Hesch et al. [85]. Those methods, even though efficient, heavily depend on linearization and Gaussian assumptions, and thus under-perform their optimization-based counterparts, such as OK-VIS [121], ORB-SLAM [140], and LSD-SLAM [52].

Graph-based SLAM typically includes two main components: the front-end and the back-end. The front-end extracts relevant information (e.g. salient features) from the sensor data and associates each measurement to a specific map feature, while the back-end performs graph optimization on a graph of abstracted data produced by the front-end.

Graph-based SLAM can be categorized either as feature-based or direct methods depending on the type of front-end. Feature-based methods rely on local features (e.g. SIFT, SURF, FAST, ORB, etc.) for pose estimation. For example, ORB-SLAM [140] performs data association and camera relocalization with ORB features and DBoW2 [67]. RANSAC [58] is commonly used for geometric verification and outlier rejection, and there are also prioritized feature matching approaches [178]. However, hand-engineered feature detector and descriptors are not robust to motion blur, illumination changes, or strong viewpoint changes, any of which can cause localization to fail.

To avoid some of the aforementioned drawbacks of feature-based approaches, direct methods, such as LSD-SLAM [52], utilize extensive photometric information from the images to determine the pose, by minimizing the photometric error between corresponding pixels. This approach is in contrast to feature-based methods, which minimize the reprojection error. However, such methods are usually not applicable to wide baseline settings [23] during large viewpoint changes. Recent work in [61] [62] combines feature and direct methods by minimizing the photometric error of features lying on intensity corners and edges. Some methods focus on dense recontruction of the scene, for instance [214] builds dense globally consistent surfel-based maps of room scale environments explored using an RGB-D camera, without pose graph optimisation, while KinectFusion [143] obtains depth measurements directly using active sensors and fuse them over time to recover high-quality surface maps. These approaches still suffer from strict calibration and synchronization requirements, and the data association modules require extensive parameter tuning in order to work correctly for a given scenario.

In light of the limitations of feature-based and direct approaches, deep networks are proposed to learn suitable feature representations that are robust against motion blur, occlusions, dynamic scenes, illumination, texture, and viewpoint changes. They have been successfully applied to several related multiview vision problems, including learning optical flow [57], depth [124], homography between frame pairs [44], and localization [26] and re-localization problems.

Recent work includes re-formulating the localization problem as a classification task [213], a regression task [106] [209], end-to-end trainable filtering [78], and differentiable RANSAC [18]. More specifically, PlaNet [213] formulates localization as a classification problem, predicting the corresponding tile from a set of tiles subdividing Earth surface for a given image, thus providing the approximate position from which a photo was taken. PoseNet [106] formulates 6-DoF pose estimation as a regression problem. One drawback of the PoseNet approach is its relative inaccuracy, compared to state-of-the-art SIFT methods. Similarly, [131] fine-tunes a pretrained classification network to estimate the relative pose between two cameras. To improve its performance, [209] added Long-Short Term Memory (LSTM) units to the fully-connected layers output, to perform structured dimensionality reduction, choosing the most useful feature correlations for the task of pose estimation. From a different angle, DSAC [18] proposes a differentiable RANSAC so that a matching function that optimizes pose quality can be learned. These approaches are not robust to repeated structure or similar looking scenes, as they ignore the sequential and graphical nature of the problem. Addressing this limitation, work in [34] fused additional sequential inertial measurement with visual odometry. SemanticFusion [130] combines convolutional neural networks (CNNs) and a dense ElasticFusion [214]. However, classic feature-based methods still outperform CNN-based methods published to date in terms of accuracies.

Recently, there has been an increasing interest in combining navigation and plannning in an end-to-end deep reinforcement learning (DRL) framework. The efforts to date can be divided into two categories depending on the presence of external memory in the architecture or not. Target-driven visual navigation takes a visual observation and an image of the target [223] or range findings [197] as input, and plans goal seeking actions in a 3D indoor simulated environment as the output.

In simulated environments, [133] uses stacked LSTM in a goal-driven RL problem with auxilary tasks of depth prediction and loop-closure classification, while [220] added successor features to ease transfer from previously mastered navigation tasks to new ones. Work in [13] augmented DRL with Faster-RCNN for object detection and SLAM (ORB-SLAM2) for pose estimation; observing images and depth from VizDoom, they built semantic maps with 3D reconstruction and bounding boxes as input to a RL policy.

To deal with the limited memory of standard recurrent architures (such as LSTM) more structured external memories have been developed to take the spatial relations of memories into account. [76] assumes known ego-motion and constructs a metric egocentric multi-scale belief map (top-down-view latent representation of free space) of the world with a 2D spatial memory, upon which RL plans a sequence of actions towards goals in the environment with a value iteration network. Neural Map in [151] is a writable structured 2D external memory map for an agent to learn to navigate within 2D and 3D maze environments. These works all assume precise egomotion and thus perfect localization, a prerequisite that can rarely be met in real-world scenarios. Relaxing this assumption and resembling traditional occupancy grid SLAM, Neural SLAM [221] uses an occupancy-grid-like memory map, assuming only an initial pose is provided, and updates the pose beliefs and grid map using end-to-end DRL.

One of key ingredient for the success of graph-based SLAM is the back-end optimization. The back-end builds the pose graph, in which two pose nodes share an edge if an odometry measurement is available between them, while a landmark and a robot-pose node share an edge if the landmark was observed from the corresponding robot pose. In pose graph optimization, the variables to be estimated are poses sampled along the trajectory of the robot, and each factor imposes a constraint on a pair of poses. Modern SLAM solvers exploit the sparse nature of the underlying factor graph and apply iterative linearization and optimization methods (e.g. nonlinear least squares via the Gauss-Newton or Levenberg-Marquardt algorithm). Several such solvers achieve excellent performance, for example, g2o [111], TSAM [43], Ceres, iSAM [98], SLAM++ [176], and recently [17] for optimization with semantic data association. The SLAM back-end offers a natural defense against data association and perceptual aliasing errors from the front-end, where similarly looking scenes, corresponding to distinct locations in the environment, would deceive place recognition. However, they depend heavily on linearization of the sensing and motion models, and require good initial guesses. Current systems can be easily induced to fail when either the motion of the robot or the environment are too challenging (e.g. fast robot dynamics or highly dynamic environments) [23].

In this work we formulate a complete end-to-end trainable solution to the graph-based SLAM problem. We present a novel architecture that combines a CNN-based local front-end and an attention-based differentiable back-end. We learn effective features automatically and perform implicit loop closure by designing an additional differentiable Neural Graph Optimizer to perform global optimization over entire pose trajectories and correct errors accumulated by the local estimation model.

4.2 Neural Graph Optimizer

The Neural Graph Optimizer architecture is split into distinct differentiable components. Similar to many of the previous SLAM methods, we split the process into local adjustments between temporally adjacent frames combined with a global optimization procedure which distributes error over the entire observed trajectory. As will be shown in the experiments, the global graph optimization procedure is critical to removing drift (the accumulation of small errors over long trajectories). The graph optimization procedure does this by learning to do loop closures, recognizing when the agent has revisted the same location, and enforcing a constraint that those poses should be nearly equal. The local model is crucial for providing a good starting point for the global optimization. It does this by estimating relative transformations between two temporally adjacent frames. By accumulating transformations from the start of the trajectory to the end, we can use this model to get the initial pose estimate within the global frame.

The complete model architecture is shown in Fig. 4.3. We will describe relative poses as $\Delta P = (\Delta p_1, \ldots, \Delta p_T)$ with the first pose set as the origin, i.e. Δp_1 is the transformation from origin to pose 1, Δp_2 is the transformation from pose 1 to pose 2, and so on. These relative poses can be transformed into a global frame of reference by accumulating the relative pose changes along the trajectory, i.e. $p_1 = \Delta p_1 \mathbf{I}$, $p_2 = \Delta p_2 \Delta p_1 \mathbf{I}$, and so on. These global poses will be refered to as $P = (p_1, \ldots, p_T)$. There exists a differential function $r2g = g2r^{-1}$ such that $P = r2g(\Delta P)$ and $\Delta P = g2r(P)$. Each component is described in more detail in the next sections.

4.2.1 Local Pose Estimation Network

The Local Pose Estimation network learns to predict the relative pose change between two consecutive frames. From two consecutive observations, where each observation is, for example, an RGB frame, this component predicts the x-coordinate, y-coordinate, and orientation (Δx, Δy and $\Delta\theta$) of the second frame with respect to the first frame. It can also optionally take in side information, such as the action taken by the agent between the two frames. The architecture of the Local Pose Estimation network is shown in Fig. 4.2. Both frames are stacked and passed through a series of convolutional layers. The output of the convolutional layers is flattened and passed to two fully-connected layers that predict the translational and rotational pose change respectively.

Some of the recent work showed that optical flow is useful in predicting frame-to-frame ego-motion [36]. The architecture of the Local Pose Estimation network is inspired by the architecture of Flownet [57] which predicts the optical flow between two frames. The convolutional layers in the Local Pose Estimation network are identical to the convolutional layers in Flownet. Prior work on visual odometry and visual inertial odometry has also used the convolutional layer architecture of Flownet [34, 211].

4.2.2 Pose Aggregation

The next step of the architecture is a Pose Aggregation network which takes in a large number of low-level poses and pose features (up to 2000 for 2D, 1000 for 3D VizDoom environment) and

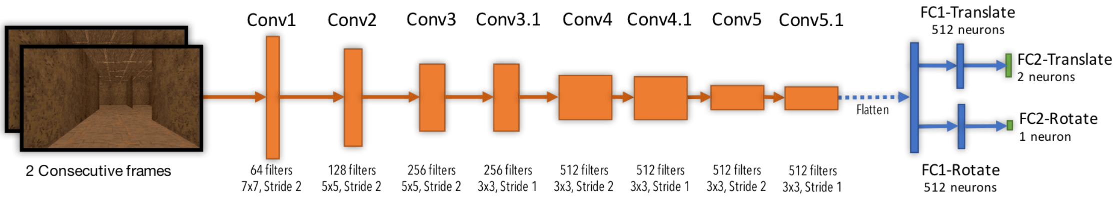

2 Consecutive frames
Conv1
64 filters 7x7, Stride 2
Conv2
128 filters 5x5, Stride 2
Conv3
256 filters 5x5, Stride 2
Conv3.1
256 filters 3x3, Stride 1
Conv4
512 filters 3x3, Stride 2
Conv4.1
512 filters 3x3, Stride 1
Conv5
512 filters 3x3, Stride 2
Conv5.1
512 filters 3x3, Stride 1
Flatten
FC1-Translate 512 neurons
FC2-Translate 2 neurons
FC2-Rotate 1 neuron
FC1-Rotate 512 neurons

Figure 4.2: The architecture of the Local Pose Estimation network. The architecture of the convolutional layers is adapted from the architecture of the Flownet [57].

reduces them into a smaller number of more temporally distant "meta-poses" and "meta-pose features" (around 250 for 2D, 125 for 3D VizDoom). These resulting meta-poses and meta-pose features are then passed to the Neural Graph Optimization procedure.

For pose feature aggregation, we utilize a deep temporal convolutional network with several alternating layers of (kernel size 3, stride 1, padding 1) dimension-preserving convolutions and (kernel size 2, stride 2, padding 0) dimension-reducing max pooling (where each max pooling operation halves the sequence size). The number of times we halve the sequence length is a hyperparameter. Instead of temporal convolutions, we could have utilized recurrent networks, but we decided to focus on convolutions for computational and memory-efficiency reasons.

In addition to the pose features being aggregated into meta-pose features by the temporal convolution, we also compose all the local pose transformations that were predicted by the Local Pose Estimation model. This composition gives us an initial global pose estimate for each of the meta-poses. The combined meta-features and meta-poses are then passed onto the Neural Graph Optimization layer for the final global pose adjustments, as shown in Fig. 4.3.

4.2.3 Neural Graph Optimization

The final component of our system is the "Neural Graph Optimizer". This submodule aggregates information over the entire pose trajectory with the goal of redistributing error to minimize drift. The Neural Graph Optimizer model is a neural analogue of the global optimization procedures commonly used in traditional state-of-the-art SLAM packages, such the g2o framework [111]. We define the Neural Graph Optimizer as a recurrent network submodule which takes as input sequential pose features and outputs a refined estimate of these poses.

In more detail, the Neural Graph Optimizer takes as input some initial T relative pose estimates (i.e. the aggregated output of the local pose estimation network) $\Delta\mathbf{P}^{(0)} = \left(\Delta p_1^{(0)}, \ldots, \Delta p_T^{(0)}\right)$ and produces two outputs for each pose:

$$\nabla\mathbf{P}^{(1)} = \left(\nabla p_1^{(1)}, \ldots, \nabla p_T^{(1)}\right), \text{ and}$$

$$\boldsymbol{\beta}^{(1)} = \left(\beta_1^{(1)}, \ldots, \beta_T^{(1)}\right).$$

New pose estimates are then constructed by performing an iterative update:

$$\Delta p_i^{(1)} = \Delta p_i^{(0)} + \beta_i^{(1)} \nabla p_i^{(1)}.$$

The Neural Graph Optimizer procedure can then be rerun on the new pose estimates $\Delta\mathbf{P}^{(1)} = (\Delta p_1^{(1)}, \ldots, \Delta p_T^{(1)})$ to produce $\Delta\mathbf{P}^{(2)} = (\Delta p_1^{(2)}, \ldots, \Delta p_T^{(2)})$, and so on. The process is repeated until some pre-specified number of iterations M has taken place. We then transform the refined relative pose estimates into the final global output: $\mathbf{P}^{(M)} = r2g(\Delta\mathbf{P}^{(M)})$.

The specific architecture of the Neural Graph Optimizer is based on two priors that are intuitively useful for pose optimization. The first prior is the notion that poses that are temporally adjacent should have similar outputs, while the second prior is that visually similar but temporally disparate poses should also have similar outputs since this provides a hint that a place has been revisited, thereby potentially enabling a loop closure-like correction of drift. We express these priors by using two architectural systems in the Neural Graph Optimizer. The first is a Transformer-like [207] attention phase where information is propagated over the entire sequence, and the second is a convolutional phase where local temporal information is aggregated.

Attention Phase

Suppose there is a meta-pose sequence of T steps, processed by the pose aggregation network into an initial set of features at each time step: $\mathbf{F}^{(0)} = (f_1^{(0)}, \dots, f_T^{(0)})$. The attention phase computes, for each pose, a soft-attention operation over the entire trajectory. This attention operation allows each pose to query information over long time spans. The attention phase takes as input the pose feature sequence $(f_1^{(i-1)}, \dots, f_T^{(i-1)})$ and produces for each time step a query vector: $(q_1^{(i-1)}, \dots, q_T^{(i-1)})$ using a fully-connected layer. Then, for each query vector $q_t^{(i-1)}$, a soft-attention operation is carried out to produce an attention vector $a_t^{(i-1)}$ as follows:

$$\begin{aligned} C_{tu} &= \langle q_t, f_u \rangle, \\ \alpha_{tu} &= \frac{C_{tu}}{\sum_{v=1}^{T} C_{tv}}, \\ a_t &= \sum_{v=1}^{T} \alpha_{tu} \odot f_u, \end{aligned}$$

where the superscripts $(i-1)$ were omitted for clarity of notation. This produces a sequence of attention vectors $(a_1^{(i-1)}, \dots, a_T^{(i-1)})$, which are passed along with $(f_1^{(i-1)}, \dots, f_T^{(i-1)})$ to the next "Optimization" phase.

Optimization

The optimization phase aggregates local temporal information by passing the pose features through several temporal convolutions and is responsible for producing the iterative adjustments: $\{\nabla p_1^{(i)}, \dots, \nabla p_T^{(i)}\}$ and $\{\beta_1^{(i)}, \dots, \beta_T^{(i)}\}$. The optimization phase proceeds as follows: First, the attention and feature vectors are concatenated into a new sequence of features:

$$\left(\begin{bmatrix} f_1^{(i-1)} \\ a_1^{(i-1)} \end{bmatrix}, \dots, \begin{bmatrix} f_T^{(i-1)} \\ a_T^{(i-1)} \end{bmatrix} \right).$$

These features are then passed through several layers of 1D convolutions h_l and activations σ_l:

$$\begin{bmatrix} \mathbf{F}^{(i)} \\ \nabla \mathbf{P}^{(i)} \\ \boldsymbol{\beta}^{(i)} \end{bmatrix} = \sigma_L \left(h_L \left(\dots\, h_1 \left(\begin{bmatrix} f_1^{(i-1)} \\ a_1^{(i-1)} \end{bmatrix} \dots \begin{bmatrix} f_T^{(i-1)} \\ a_T^{(i-1)} \end{bmatrix} \right) \dots \right) \right)$$

to produce the current iteration's adjustments ($\nabla \mathbf{P}^{(i)}$ and $\boldsymbol{\beta}^{(i)}$) as well as the feature layer for the next iteration of the process ($\mathbf{F}^{(i)}$).

For our experiments, we use 9 layers of convolutions with filter size 3 and ReLU activations. While temporal convolutions have a limited receptive field which provides a hard upper limit on how far they can transmit information across time, we found that in practice they worked better than using a bidirectional LSTMs.

Induced Attention Graph

We now provide some intuition on why the attention phase enables higher performance than only using the optimization phase, or running all pose features through bidirectional LSTMs. We can see that during the attention phase, some similarity graph C is constructed such that each element C_{tu} is the inner product between the query vector q_t and the pose feature vector f_u. Therefore C represents a similarity matrix between the queries and pose features, and those with very similar features will thus have high information bandwidth through the attention operator because the attention weight α_{tu} will be near 1 for highly similar query and pose features, and near 0 otherwise. The attention operation is thus inducing a connectivity graph between poses with highly similar features. This therefore resembles a soft, differentiable analogue of the pose graph constructed in SLAM algorithms such as ORB-SLAM [140].

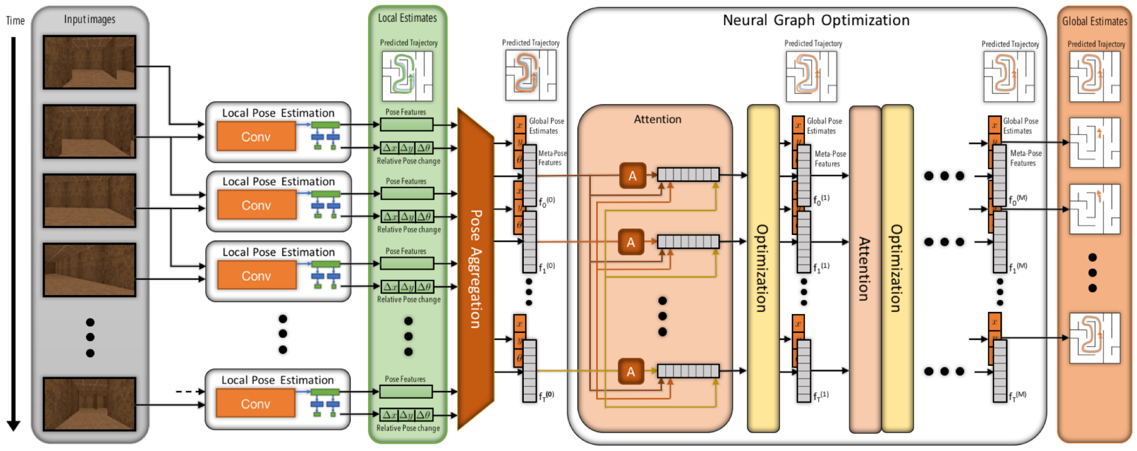


Figure 4.3: The architecture of the proposed model, showing the Local Pose Estimation, the Pose Aggregation, and the Neural Graph Optimization modules.

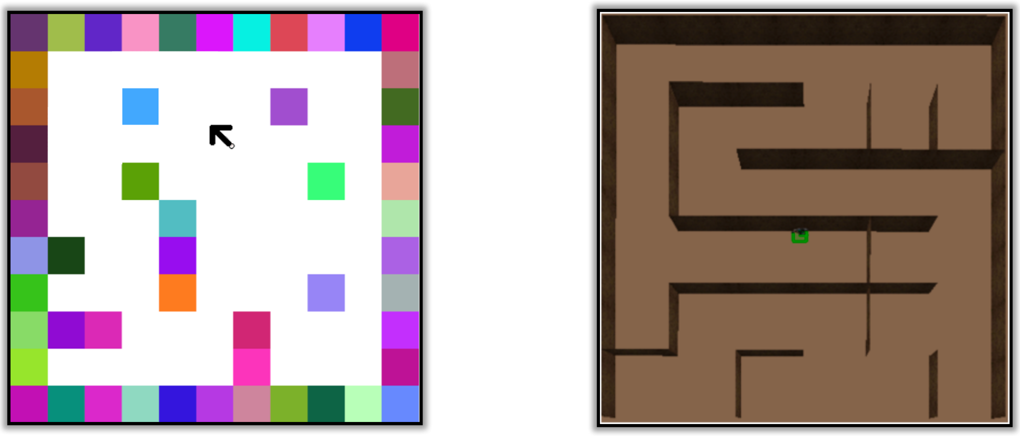


Figure 4.4: **Left:** A screenshot of the 2D environment based on Box2D. **Right:** A bird's eye view of the 3D environment based on the Doom game engine.

Results on the 2D Environment	
Model	**RMSE**
Only Local Estimation	17.80
Global Estimation - 1 Attend-Opt iteration	10.21
Global Estimation - 5 Attend-Opt iterations	3.16

Table 4.1: Results for different Neural Graph Optimizer architectures and hyperparameters, in terms of test set Global RMSE. We can see that the addition of the global optimization procedure reduces the loss by more than 80% as compared to solely using the local pose model.

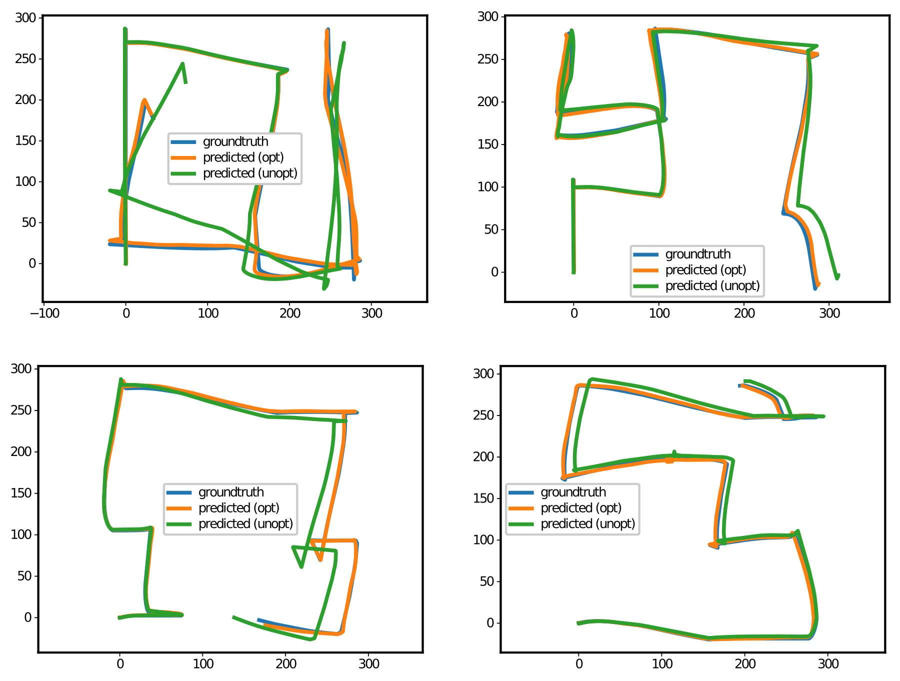


Figure 4.5: Images visually demonstrating the effect on pose estimates of adding the Neural Graph Optimizer module on top of the local pose estimation model in the 2D environment. We can see that the global optimization procedure greatly reduces drift. These figures were generated with the 5 iteration Neural Graph Optimization model.

4.3 Experiments

We use two simulation environments for our experiments, a 2D environment based on Box2D and a 3D environment based on the Doom game engine. To train the system, we pretrained the local pose estimation model and then trained the global optimizer with the local pose model held fixed. This was mainly due to the large sequence lengths we were required to process (on the order of 1000 time steps). This limited the amount of sequences we could process due to the large memory requirements. Training the system in stages enabled us to preprocess the sequence images into a far more memory-efficient compressed representation.

4.3.1 2D Environment

For the 2D Environment, random maze designs are generated using Prim's algorithm [160], and the environment is created using Box2D (box2d.org). The agent projects 241 rays uniformly in front of itself with an effective field of view of 300°. The observation of the agent includes the RGB values as well as the depth of the points where these rays hit a wall. An example of the

Results on the 3D Doom Environment				
Model	**Seen**		**Unseen**	
	% Err. Trans.	% Err. Rot.	% Err. Trans.	% Err. Rot.
Only Local Estimation	1.65	0.117	1.62	0.122
Global Estimation				
- 1 Attend-Opt iteration	1.42	0.071	1.16	0.071
- 5 Attend-Opt iterations	1.25	0.057	1.04	0.056
DeepVO [211]	1.78	0.079	2.39	0.091

Table 4.2: Results for different Neural Graph Optimizer architectures and hyperparameters, in terms of % translation and rotation error on maps either seen or unseen during training time. We can see that the addition of the global optimization procedure reduces error significantly compared to using only the local pose model. In addition, increasing the number of attention iterations provides an increase in performance.

2D environment is shown in Fig. 4.4. Each cell in the maze has a random color. The agent can take one of three discrete actions at every time step: move-forward, turn-left, or turn-right. These actions result in translational acceleration if the action is move-forward or angular acceleration if the action is turn-left or turn-right. Data is collected by visiting four different corners on the maze using Dijkstra's algorithm [46].

For this environment, the training data is generated by worker threads in parallel with the model training and each training datapoint is used only once. A test set is fixed and common for all experiments. Each epoch of training consists of $200,000$ datapoints. The error metric is Root Mean Squared Error (RMSE) in pose estimation.

To improve upon the results produced by the local pose estimation model, we train a Neural Graph Optimizer on the pose outputs of a pretrained Local Pose Estimation model. For the 2D environment, as shown in Table 4.1, we observed over 80% improvement in the correction of drift compared to using only the local pose estimation model, as measured by the root mean squared error loss. We can see that increasing the number of iterations (applying the attention operator and then the temporal aggregation operator) improved results from 1 to 5 iterations. We show some sample trajectories in Fig. 4.5 before and after the Neural Graph Optimizer procedure.

4.3.2 3D Environment

For the 3D Environment, random maze designs are generated using the Kruskal's algorithm [110], and the environment is created using the ViZDoom API [105]. The agent observes the environment in a first-person view with a field-of-view of 108°. An example of the 3D environment design is shown in Fig. 4.4. Similr to the 2D environment, the pose predictions are 3-dimensional tuples (x, y, angle) and the agent can take one of three discrete actions at every time step: move-forward, turn-left, or turn-right, which results in translational or angular acceleration. For collecting data in this environment, a navigation network [114] is trained to maximize the distance travelled by the agent using the Asynchronous Advantage Actor-Critic algorithm [136]. The data is collected by using the policy learned by the navigation network.

Like the 2D environment, the training data is generated by worker threads in parallel with the model training, and each training datapoint is used only once. We additionally sample two test sets, one containing 39 trajectories sampled from maze geometries that were seen during training and one containing 39 trajectories sampled from novel maze geometries that the agent had not encountered during training.

Results

Results are shown in Table 4.2. Here we report % Error in Translation and Rotation for seen/unseen mazes, where the accumulated drift error is divided by the entire distance traveled in each trajectory. Observe that the local model is significantly improved by using global optimization and performance of the global model improves as we increase the number of Attend-Opt iterations from 1 to 5. The global model outperforms the DeepVO [211] baseline on both the test sets. Additionally, we can clearly see that the model itself does not overfit to the training environments it experienced, and gets similar or even lower error on unseen test mazes. Learning curves are shown in Fig 4.6. We can see that performance plateaus decrease significantly early on and then progress is much slower after around 2000 updates.

The baseline DeepVO [211] is one of the state-of-the-art methods using deep neural nets for monocular visual odometry. It stacks 2 consecutive frames and passes them through 9 convolutional layers followed by 2 LSTM layers to estimate the pose changes. As compared to the proposed Local Pose Estimation model which observes only the last 2 frames at the time, the DeepVO model can potentially utilize information from all the prior frames using the LSTM layer. However, the DeepVO model does not correct its previous predictions as it observes new information. The Neural Graph Optimizer has the ability to correct its predictions using the Attention operation and consequently leads to improved performance.

Analysis

We next plot the total rotational and translational errors as a function of number of steps in the trajectory in Figures 4.7 (for unseen mazes) and 4.8 (for seen mazes). The global model reduces the slope of the rate of increase of both translational and rotation errors as compared to the local

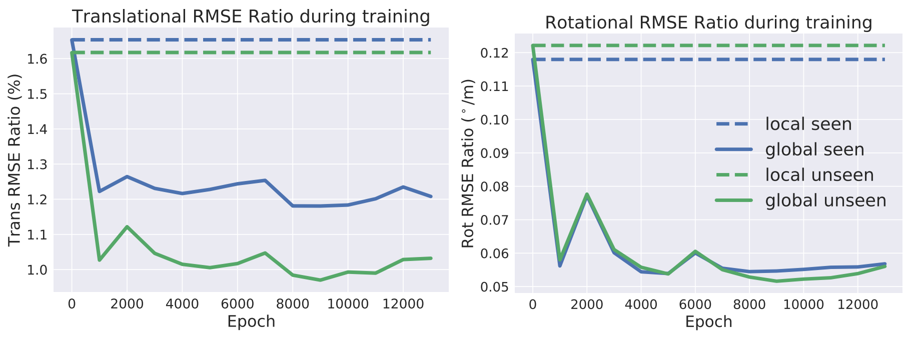


Figure 4.6: Training curves for Doom over 13,000 updates for the 5 iteration Attend-Opt model. We show the performance on both seen and unseen test sets as training progress. The dotted line represents the estimate provided by using only the local model. We can see there is a large reduction in error when making use of the global optimizer.

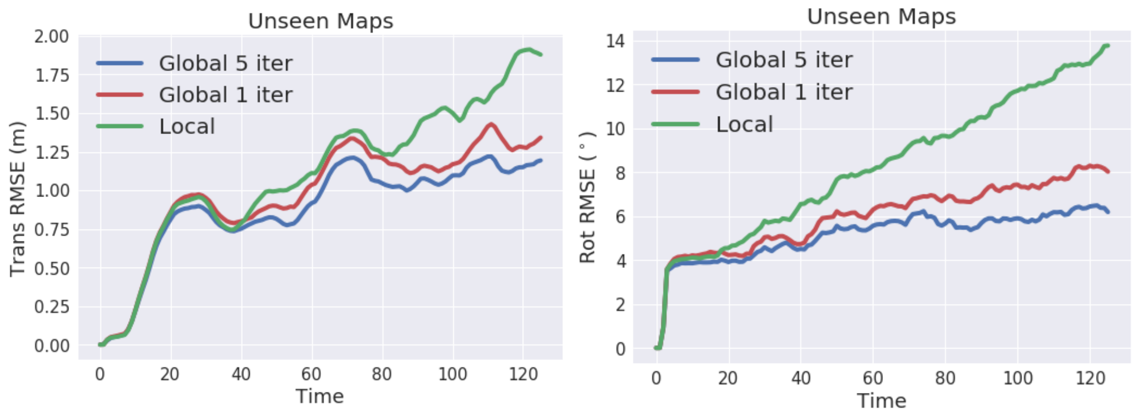


Figure 4.7: Translational (Left) and Rotational (Right) RMSE as a function of number of images in the trajectory in **unseen mazes**.

estimates. Figures 4.9 and 4.10 display the ratio of the translational (left) and rotational (right) drift error over distance traveled. We can see from these plots that the trend is negative, meaning that drift accumulates much slower than the distance being traveled. This indicates that the model is likely to generalize well to arbitrarily long trajectories. Additionally, in all plots, we can see a clear ordering of the performance of the models, where the local model performs worst, one iteration of Attend-Opt increases model performance significantly, and increasing the number of Attend-Opt iterations to 5 further increases model performance.

The plots in Figures 4.7 and 4.8 as well as the numbers in Table 4.2 show that the improvement in rotational errors due to the neural optimization is higher than the improvement in translation errors. Fig 4.11 shows sample trajectories with estimates of both global and local pose estimates. As seen in the figure, the neural graph optimizer considerably improves the rotation estimates, consequently leading to significant improvements in the drift reduction.

4.4 Discussion

In this chapter, we designed a novel attention-based architecture to perform an end-to-end trainable global pose estimation. Compared to the previous work on using deep networks to do pose estimation, our method uses an attention operation to re-estimate its trajectory at each time step and therefore enables iterative refinement of the quality of its estimates as more data is available. We demonstrate the benefit of the model on two simulators, the first is a top-down 2D maze world and the second is a 3D random maze environment running the Doom engine. Our results show that our method has an increased performance compared to models which used only temporally local information.

With the localization method designed and tested, in this final chapter of this part of the thesis we have now assembled all the components necessary for a monolithic agent architecture capable of

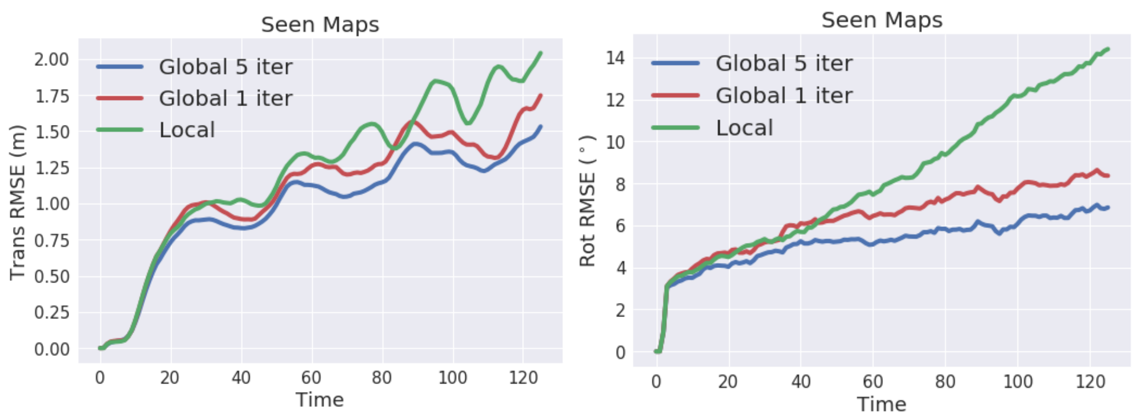


Figure 4.8: Translational (Left) and Rotational (Right) RMSE as a function of number of images in the trajectory in **seen mazes**.

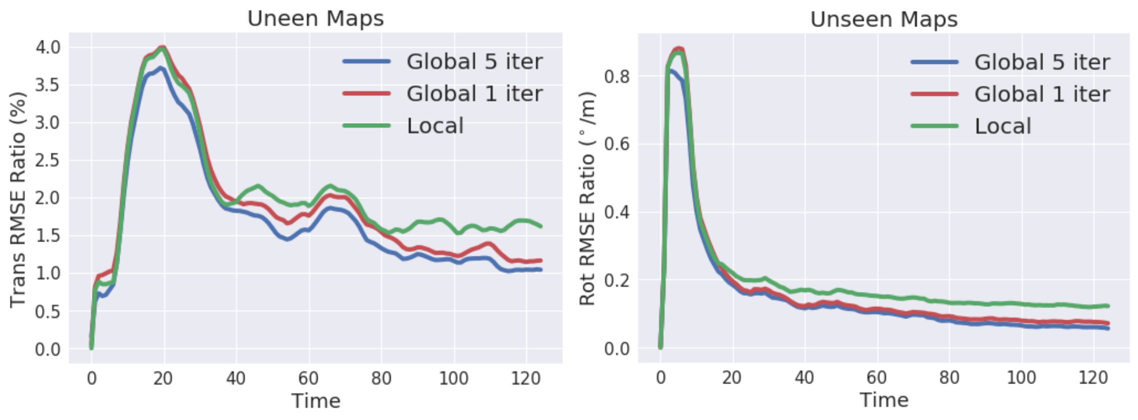


Figure 4.9: Ratio of the Translational (Left) and Rotational (Right) RMSE to the distance travelled as a function of number of images in the trajectory in **unseeen mazes**.

rapid adaptation in embodied environments:

- The Neural Map, an inter-episodic memory that leverages geometric structure to spatially organize observed feature representations into a "map" representation. We demonstrated that this map enabled learning few-shot navigation policies more rapidly than less structured baselines like LSTMs and memory networks.
- The Gated Path Planning Network (GPPN), a differentiable submodule capable of state-of-the-art path planning for embodied environments. The GPPN does not require any pre-specified MDP model information and can do planning with any distributed map representation, including those produced by the Neural Map. We demonstrated that the GPPN achieves superior performance, stability and learning speed compared to the previous state-of-the-art.
- Active Neural Localization, a differentiable submodule capable of re-localization in an environment given a map, enabling the agent to rapidly disambiguate its location. The Active Neural Localization submodule works with any distributed map, including those produced by the Neural Map. Furthermore, we defined an auxiliary reward objective that encouraged the agent to choose information-seeking actions, reducing its uncertainty even more rapidly.
- The Neural Graph Optimizer presented in this chapter, a differentiable submodule enabling localization without any map or location estimate available *a priori*. The Neural Graph Optimizer can plug in as the submodule providing allocentric location or egomotion estimate to the Neural Map.

Combined, these submodules can create an agent that would have the capabilities of state-of-the-art differentiable memorization, planning, and state inference, and which can perform all methods

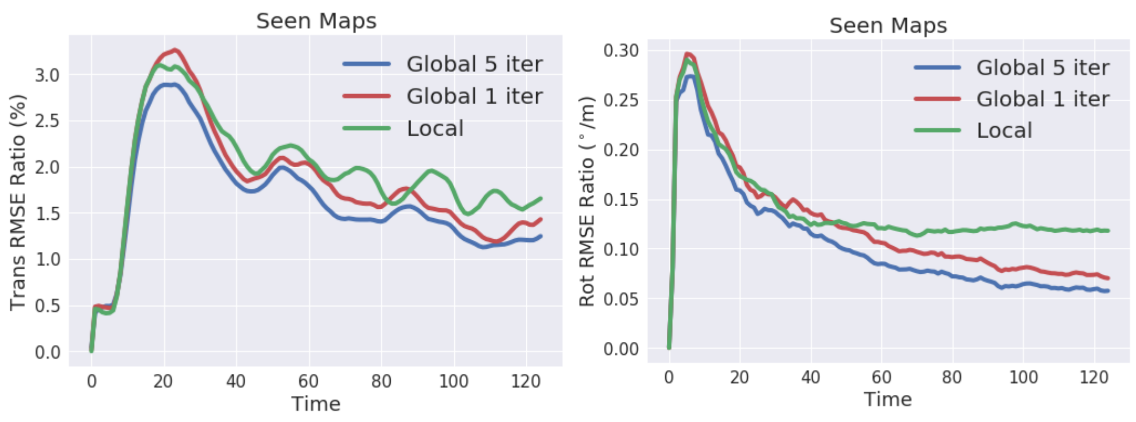


Figure 4.10: Ratio of the Translational (Left) and Rotational (Right) RMSE to the distance travelled as a function of number of images in the trajectory in **seen mazes**.

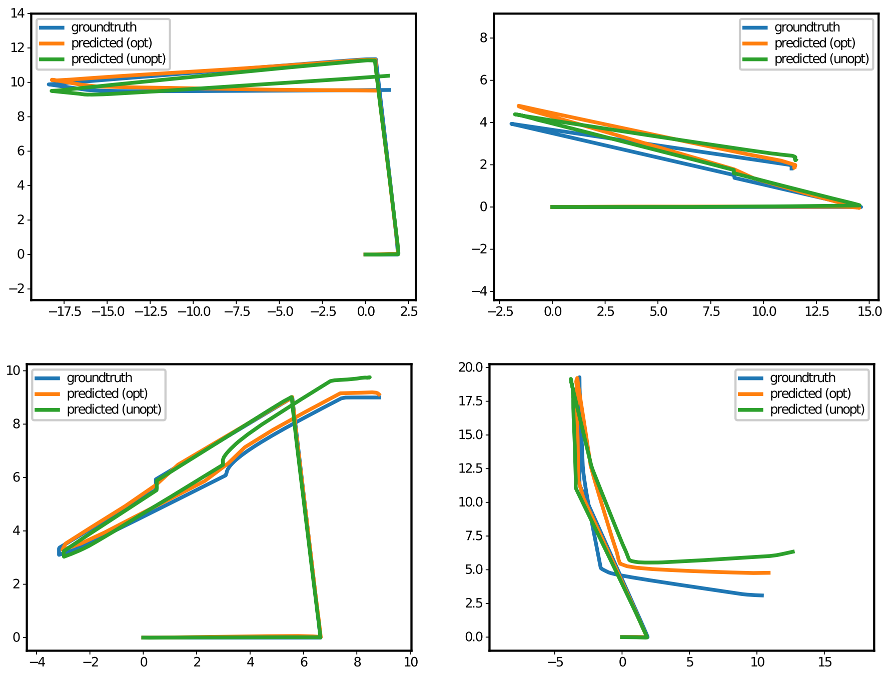


Figure 4.11: Images visually demonstrating the effect on pose estimates of adding the Neural Graph Optimizer module on top of the local pose estimation model in the 3D environment. We can see that the global optimization procedure greatly reduces drift. These figures were generated with the 5 iteration Neural Graph Optimization model. The agent always starts at the origin (0, 0).

simultaneously. In the next part of the thesis, we move on to focus on meta-learning through inter-episodic memory in general environments, a less structured setting requiring less domain-specific inductive biases. To start, in the next chapter we will question one of the most central facets of the meta-learning methods considered so far: how the agent interacts with an environment.

Part II

Meta Reinforcement Learning through Memory

Chapter 5

Concurrent Episodic Meta Reinforcement Learning

In the previous part of this thesis we focused on meta-learning in embodied environments, working towards the subcomponents necessary to build a monolithic embodied agent architecture that could be capable of simultaneously performing memorization, planning, and state inference. The components we designed relied heavily on making use of biases present in the environment class: that the state-space possessed underlying euclidean geometric structure, that the agent actions caused only local movement, that observations were projections of 3D objects located in the scene, etc. While an important domain, many applications do not fall under this category of environments and would not benefit from the use of the components we designed. Therefore in this second part of the thesis we now focus on meta-learning through inter-episodic memory in general environments, with no constraints on the MDPs we consider for rapid adaptation. This means less structure is available and accordingly more general priors will need to be exploited that make use of less domain-specific knowledge.

In this chapter, we specifically re-examine how a meta-learning agent interacts with its environment. Typically in the context of meta-learning with inter-episodic memory, a single agent continuously interacts with a particular environment instance: meta-episodes comprised of a *sequence* of interaction episodes with the same environment instance, and the next action is chosen by the agent through conditioning on the inter-episodic memory comprising statistics from the sum total of the interaction history with that instance. A negative side-effect of this sequential execution paradigm is that, as the environment becomes more challenging, and thus requiring more interaction episodes, it needs the agent to reason over longer time-scales. These methods also intrinsically suffer from risk-aversion or reward sparsity due to their learning objectives, as will be evidenced in this chapter. To combat these difficulties we propose an alternative parallel framework, which we name "Concurrent Episodic Meta-Reinforcement Learning" (CEMRL), that transforms the temporal credit assignment problem into a multi-agent one. In CEMRL, a set of parallel agents are executed in separate, identical task instances, with each agent given the means to communicate with any other. The goal of the communication is to coordinate, in a collaborative manner, the most efficient exploration of the shared task instance.

This coordination therefore represents the meta-learning aspect of the framework, as each agent can be assigned or assign itself a particular section of the current task's state space. This transforms the setting of a single agent reading and writing from a single memory, into one of multiple agents reading from and writing into a single shared memory. The shared memory, acting as a stateful communication channel, replaces the inter-episodic memory as the main mechanism driving rapid adaptation. CEMRL stands in contrast to standard RL methods that assume that each parallel rollout occurs independently, which can potentially waste computation if many of the rollouts end up sampling the same part of the state space. Furthermore, the parallel setting enables us to define several reward sharing functions and auxiliary losses that are non-trivial to apply in the sequential setting. We demonstrate the effectiveness of our proposed CEMRL at improving over sequential methods in a variety of sparse, high-risk tasks.

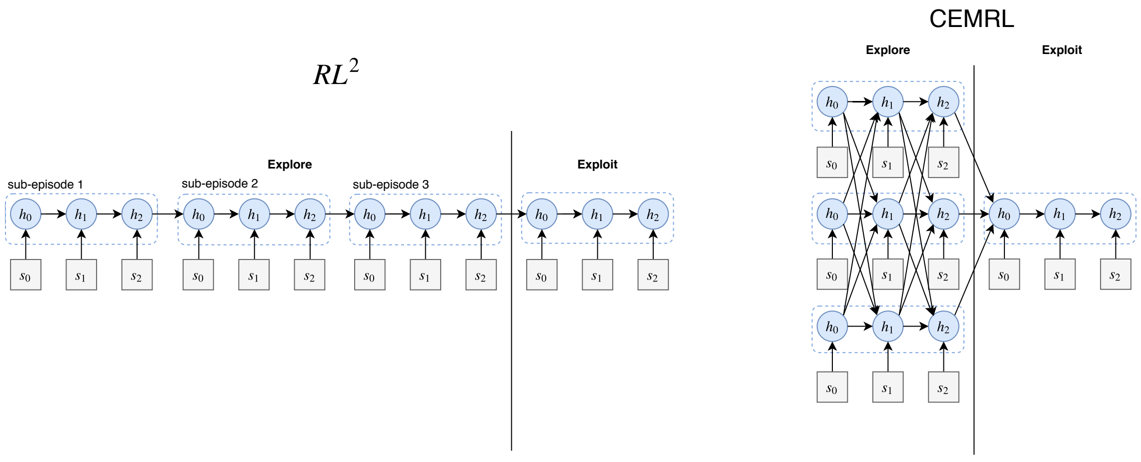


Figure 5.1: Our proposed parallel CEMRL framework (right) and the sequential RL2 setting (left).

5.1 Background & Related Work

In this section, we standardize the episodic meta-learning algorithms within a common framework. We assume in the environments we test on that there are K episodes in a meta-episode, and we further explicitly sub-divide these episodes into two types of categories. The first type is the explore episodes, of which there are $K_{explore}$. The second type is the exploit episodes, which there are $K_{exploit}$. We have that $K = K_{explore} + K_{exploit}$. In this chapter, we will only consider settings in which $K_{exploit} = 1$.

In contrast to previous work on inter-episodic memory, in this chapter we use separate RNN weights and policy outputs between explore and exploit sub-episodes. This makes RL2 more closely resemble a sequence-to-sequence model [195], as it first encodes exploratory sub-episodes into a vector and then uses that code vector as the initial state of an exploitative recurrent policy. While a minor change, the separation of explore and exploit policy parameters can enable an easier optimization for the meta-learner to learn to produce different behaviour between explore and exploit sub-episodes.

Concurrent Reinforcement Learning: A related work in the non-meta-learning setting is that of Concurrent Reinforcement Learning (CRL) [185]. The CRL setting is defined as the case where episodic interactions occur in parallel, and [185] designed and analyzed a temporal difference update to handle learning in this setting. More recent work [48, 47] in the concurrent setting applied techniques from the Posterior Sampling for Reinforcement Learning (PSRL) literature [192].

5.2 Concurrent Episodic Meta Reinforcement Learning

Episodic Meta-RL algorithms process episodes in a sequential manner, conditioning on all past history to influence the behaviour within the next interaction episode. While the sequential processing of episodes has been shown to be effective [49, 210], it has several drawbacks including long meta-horizons complicating optimization, and risk-adverseness / reward sparsity. As a concrete example highlighting these latter drawbacks, consider a class of MDPs with 1 initial state and 4 absorbing states (Fig. 5.3). The initial state can transition to any absorbing state: transitioning to three of the four absorbing states provides a reward (termed goal states), the other provides no reward (termed NOOP). Suppose that within this class of MDPs (which we call Monty-Hall), each MDP is defined as having a randomly chosen goal such that moving to it from the initial state results in a reward of $+0.1$, and transitioning to any other goal results in a reward of -1.

In this environment, RL2 [49, 210] will try to naively maximize its cumulative reward over every sub-episode within a meta-episode, eventually learning not to enter any goal. This is because entering an incorrect goal incurs a large negative reward of -1, a loss from which RL2 will not recover by finding out which goal has $+0.1$ reward, whereas transitioning to the NOOP state has value 0. RL2 is thus not capable of producing optimal behaviour within certain classes of MDPs, especially ones where exploration has a significant risk of negative reward. ERL2 [190] addressed some of these concerns by splitting the meta-episode into clearly delineated exploratory and exploitative

Algorithm 1 Concurrent Episodic Meta-RL

```
# Explore Phase
$\forall.k \quad s_0^{(k)} = env^{(k)}.reset()$
for $t$ in $[1, T]$ do
    for $k$ in $K_{explore}$ do
        $a_t^{(k)} \sim \pi^{(k)}(\cdot | s_{t-1}^{(k)}, h_{t-1}^{(k)}, h_{t-1}^{m})$
        $s_t^{(k)}, r_t^{(k)} = env^{(k)}.step(a_t^{(k)})$
        $h_t^{(k)} = f^{(k)}(s_t^{(k)}, a_t^{(k)}, r_t^{(k)}; h_{t-1}^{(k)}, h_{t-1}^{m})$
                                                    ▷ Per-Rollout Update
    end for
    $h_t^m = g(h_t^{(1)}, \ldots, h_t^{(K_{explore})}; h_{t-1}^m)$      ▷ Meta Update
end for
# Exploit Phase
$s_0 = env.reset()$
$h_0^{exploit} = r(h_T^m)$
for $t$ in $[1, T]$ do
    $a_t \sim \pi^{(k)}(\cdot | s_{t-1}, h_{t-1}^{exploit})$
    $s_t, r_t = env.step(a_t)$
    $h_t^{exploit} = f^{exploit}(s_t; h_{t-1}^{exploit})$
end for
```

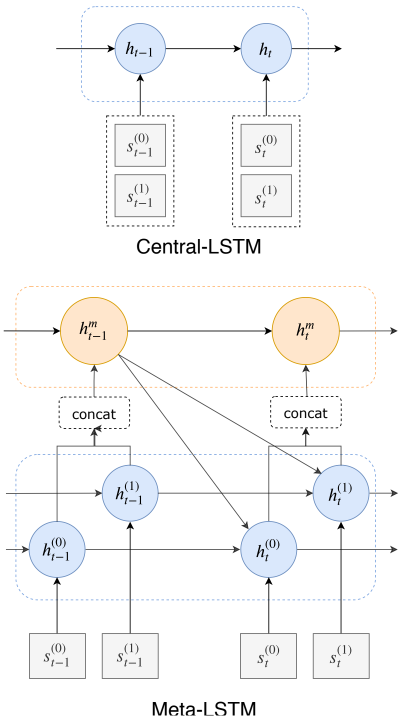


Figure 5.2: **Left:** Pseudo-code for CEMRL. Environment steps are processed in parallel. There are distinct per-rollout hidden states (processing single-agent trajectories) and meta hidden states (aggregating information from all rollouts). The meta hidden state is used as a conditioning vector for an exploit policy, which is trained to maximize reward. **Top-Right:** Central-LSTM is a monolithic LSTM that treats the problem as a single agent task with a factored output distribution, concatenating all information into one combinatorial input space. **Bottom-Right:** Meta-LSTM is a structured memory-based communication architecture where each rollout has a separate LSTM (with weights shared between rollouts). The rollouts communicate through the Meta-LSTM.

sub-episodes, and trained the policy output using only the external reward accumulated on exploit sub-episodes. This enables high-risk exploration but comes at the cost of an even more difficult temporal credit assignment problem as the external reward signal is sparsified when dealing with a large number of interactions.

To combat the aforementioned issues that sequential meta reinforcement learning methods suffer from, we propose a parallel framework termed "Concurrent Episodic Meta Reinforcement Learning" (CEMRL). CEMRL transforms the step-by-step processing of interaction episodes into a parallel processing of all episodes concurrently, reducing the meta-horizon to the time horizon of the environment. These parallel rollouts are executed by separate agents capable of communication with each other, enabling the coordinated assignment of specific agents to particular sections of the state space for efficient exploration. The communication protocol between rollouts therefore represents the meta-learning process, where agents transmit information about rewards, dynamics or states to each other and aggregate this information to present to the exploitation sub-episode policies in order for them to act optimally. Fig. 5.1 visually demonstrates the main differences between RL2 and CEMRL.

By this conversion of the temporal credit assignment problem into a multi-agent communication learning problem, we are afforded other advantages beyond a reduced time horizon that no longer scales with the number of interactions. The first is the use of specialized reward sharing schemes, which can be used to re-distribute risk among the exploratory rollout policies in a way not straightforwardly possible within the sequential setting. Additionally, we can define divergence losses between rollout policies to encourage exploration and diversity of agent behaviours.

We present the general algorithmic description of our CEMRL framework in Algorithm 1. Environments are processed in parallel, step by step, by a two-level hierarchy. The lower-level per-rollout representations $h^{(k)}$ are responsible for processing information from a single rollout. The high-level meta representation h^m aggregates all the per-rollout hidden states at every time step.

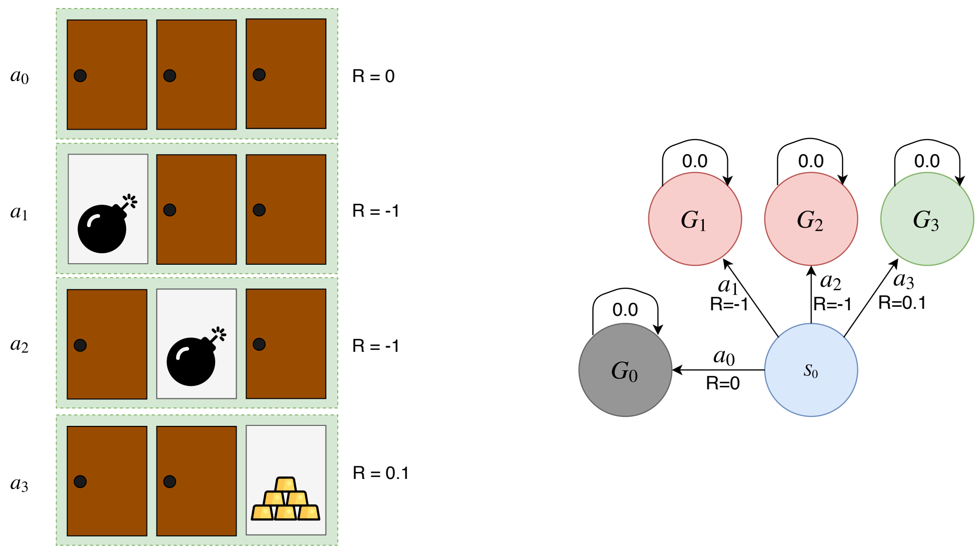


Figure 5.3: An example illustration (**Left**) and its MDP (**Right**) of a particular instance environment of the Monty-Hall class of MDPs. From the starting state s_0, the agent is given a choice of 4 actions. The first action is a NOOP and the action ends the episode without reward. The other actions open one of the 3 doors, where 2 of the doors lead to large negative rewards (illustrated with a cartoon bomb) and one door leads to a positive reward (represented by the gold bars).

Meta-learning is achieved through the conditioning of the per-rollout networks $f^{(k)}$ on the previous time-step's h^m, and the aggregation of all per-rollout hidden states $h^{(k)}$ into h^m. We now discuss the particular architectures of per-rollout update $f^{(k)}$ and meta update g we used for experiments.

5.2.1 Communication Architecture

The meta-learning component of the algorithm is enabled by the communication protocol between rollout agents, and therefore the design of this protocol's architecture is critical to the meta-learner's performance. Recent work has seen a large variety of architectures developed to enable communication between agents in the setting of deep multiagent reinforcement learning [193, 60, 90]. These various methods allow communication between agents using either discrete protocols [60] or continuous ones [193, 60], and they have been shown to be effective at solving multiagent tasks. While CEMRL does not prescribe a particular method of inter-rollout communication, we describe the architectures utilized in the experiments below, with a visual depiction given in Fig. 5.2.

Central-LSTM: The first communication architecture is an LSTM which aggregates all rollout information (states, rewards, etc.) into a single monolithic input. The Central-LSTM thus processes the multiagent data as if it were from a single-agent task with a combinatorial input space and a factored output distribution. Specifically, we define:

$$\begin{aligned} f^{(k)}(s_t^{(k)}, a_t^{(k)}, r_t^{(k)}; h_{t-1}^{(k)}, h_{t-1}^m) &= [s_t^{(k)}, a_t^{(k)}, r_t^{(k)}] \\ g(h_t^{(1:K_{explore})}; h_{t-1}^m) &= \text{LSTM}([h_t^{(1:K_{explore})}]; h_{t-1}^m) \end{aligned}$$

Meta-LSTM: The second communication architecture considered has more structure suited to the particular application of CEMRL. In this architecture, each rollout has a separate "Rollout-LSTM" with shared recurrent and input weights. Communication occurs through a stateful communication channel represented by a centralized "Meta-LSTM" that aggregates information from all the separate Rollout-LSTMs, and then sends its features on the next time-step to each Rollout-LSTM. Formally:

$$\begin{aligned} x_t^{(k)} &= [s_t^{(k)}, a_t^{(k)}, r_t^{(k)}, h_{t-1}^m] \\ f^{(k)}(s_t^{(k)}, a_t^{(k)}, r_t^{(k)}; h_{t-1}^{(k)}, h_{t-1}^m) &= \text{LSTM}(x_t^{(k)}; h_{t-1}^{(k)}) \\ g(h_t^{(1:K_{explore})}; h_{t-1}^m) &= \text{LSTM}([h_t^{(1:K_{explore})}]; h_{t-1}^m) \end{aligned}$$

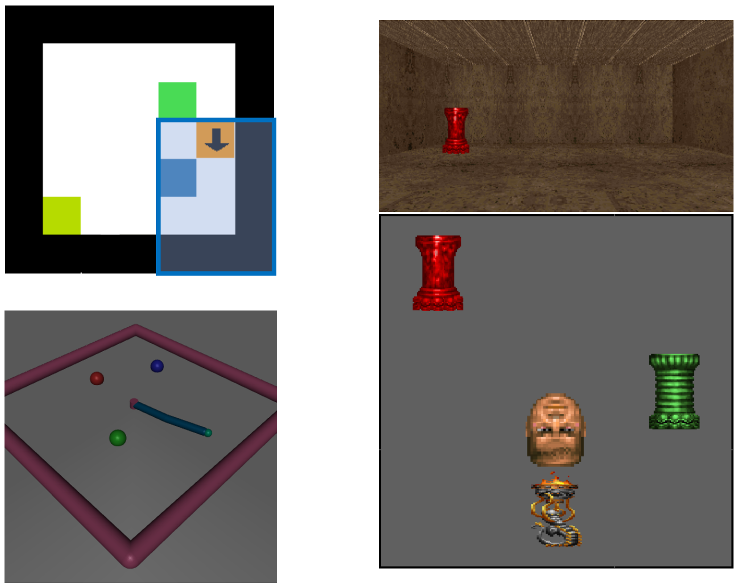


Figure 5.4: **Top Left:** An example of a 3-Color-Choice environment with 3 different colored goals. The blue rectangle represents the agent's current field of view (pixels used as its state). The goal of the environment is to navigate the room, and determine which of the colored goals contains the positive reward. **Bottom Left:** An example of the 3-Reacher environment, with three different target end-effector positions shown as colored spheres. The agent does not know which position has the positive reward beforehand and must query the goal states during its exploration phase. **Top Right:** Example RGB observation of the 3-Doom-Choice environment. **Bottom Right:** Top-down view of the state of the observation above it, rendered to show the 3 objects that the agent can collect in the room. For visualization purposes, the agent only has access to RGBD images. Distances not to scale in top-down view.

5.2.2 Reward Sharing Schemes

Parallel rollouts allow the use of a variety of reward sharing schemes. An effective reward sharing scheme is one which can redistribute risk amongst the rollouts but still allow external signals to shape the training of the policy during exploration sub-episodes. This is in contrast to the scheme used in ERL2 which completely disregards exploratory rewards during policy optimization, sparsifying the reward signal. CEMRL affords more flexibility in the design of these schemes than the sequential framework, because in the sequential case there is a significant asymmetry in the exploratory sub-episodes (sub-episode 3 is conditioned on more information than 1 and 2, etc.).

Max-Until-Exploit: Given $K_{explore}$ rollouts executing in parallel, the Max-Until-Exploit scheme takes as return the maximum over all rollout returns during explore sub-episodes, summed with the return of the exploit sub-episode. Thus during the parallel exploration sub-episodes, the Max-Until-Exploit scheme will cause agents to try to maximize at least one rollout's return, while the other rollouts are free to execute high-risk exploratory behaviours. This has the benefit of ERL2's risk-robustness while enabling an external signal on the performance of the exploration sub-episodes. The return used to train the explore policies is:

$$R_{Max} = \sum_{t=1}^{T} r_t + \max_{(k)} \sum_{t=1}^{T} r_t^{(k)}$$

StDev-Until-Exploit: The Standard Deviation(StDev)-Until-Exploit scheme takes as return the standard deviation of per-rollout returns during explore sub-episodes, summed with the return of the exploit sub-episode. The StDev-Until-Exploit scheme therefore gets higher reward when all agents produce behaviours which result in a variety of returns. While this reward sharing scheme will not be aligned to every task's objective, we found it to provide some useful learning signal in

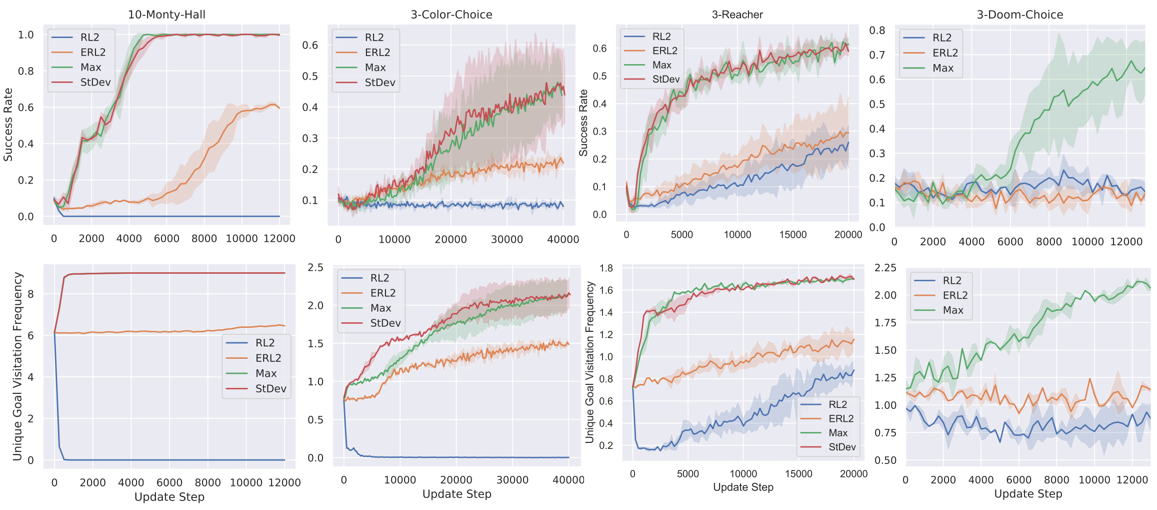


Figure 5.5: Learning curves of the models on the four environments. **Top:** success rate of the model over epochs. **Bottom:** average unique goal visitation frequency during explore sub-episodes. 3-Doom-Choice data points averaged over 160 meta-episodes & 2 seeds. All other environment data points are averaged over 1280 meta-episodes & 3 seeds per model.

very high-risk environments. More specifically, the return is:

$$R_{StDev} = \sum_{t=1}^{T} r_t + std\left(\left[\sum_{t=1}^{T} r_t^{(1)}, \dots, \sum_{t=1}^{T} r_t^{(K_{explore})}\right]\right)$$

5.2.3 Divergence Losses

Beyond reward sharing schemes, we develop several auxiliary losses which can help produce varied behaviours in the population of agents. The goal of these auxiliary losses is to drive agents into different partitions of the state space, encouraging exploration to new areas of the environments. Given that we potentially want to deal with partially-observable environments, we develop a framework that calculates divergences while still maintaining the proper memory states of each rollout. First, let:

$$\begin{aligned} h_1^{(i,j)} &= f^{(i)}(s_1^{(j)}, a_1^{(j)}, r_1^{(j)}; h_0^{(i)}, h_0^{m^{(i,j)}}) \\ h_t^{(i,j)} &= f^{(i)}(s_1^{(j)}, a_1^{(j)}, r_1^{(j)}; h_{t-1}^{(i,j)}, h_{t-1}^{m^{(i,j)}}) \\ h_t^{m^{(i,j)}} &= g(h_t^{1,\rho(1)}, \dots, h_t^{i,\rho(i)}, \dots, h_t^{K_{explore},\rho(K_{explore})}; h_{t-1}^{m}) \end{aligned}$$

Which runs rollout (i)'s architecture on rollout (j)'s trajectory and the meta architecture on some permutation ρ of the trajectories such that $\rho(i) = j$ holds. Then the divergence loss $L_D(i, j)$ is equal to:

$\sum_{t=0}^{H-1} D(\pi^{(i)}(\cdot|s_t^{(j)}, h_t^{(i,j)}, h_t^{m^{(i,j)}}), \pi^{(j)}(\cdot|s_t^{(j)}, h_t^{(j)}, h_t^{m}))$

This loss encourages different rollout policies to output distinct action probabilities on the same state history input, thereby achieving more diverse behaviours. Given the combinatorial number of permutations, we implement this efficiently in the following manner. We first cache the values of $\pi^{(i)}(\cdot|s_{t-1}^{(i)}, h_{t-1}^{(i)}, h_{t-1}^{m})$ obtained during the policy rollout, then sample a permutation ρ and rollout all K agents simultaneously on the permuted trajectories to calculate the sum over i of $L_D(i, \rho(i))$. To further improve efficiency, we restrict the permutations we sample to derangements (where no element appears in its original position). We consider two forms of symmetric divergences: the symmetric KL Divergence and the JS Divergence.

5.3 Experiments

We validate CEMRL on four environments N-Monty-Hall, N-Color-Choice, N-Reacher and N-Doom-Choice, comparing against the baseline algorithms RL2 and ERL2. Results for the environments are

10-Monty-Hall								
Model	AuC	Final Perf.	Updates until Success Rate					Visited Goals
			25%	50%	75%	95%	100%	
RL2	19.83	0.00 ± 0.0%	-	-	-	-	-	0.00 ± 0.0
ERL2	2939.17	59.5 ± 0.1%	7500	9250	-	-	-	6.42 ± 0.0
Max-Until-Exploit	**9575.42**	**100 ± 0.0%**	1250	2750	3750	4500	5000	**9.00 ± 0.0**
StDev-Until-Exploit	**9501.67**	**99.7 ± 0.5%**	1250	2500	3750	5000	6500	**9.00 ± 0.0**

3-Color-Choice							
Model	AuC	Final Perf.	Updates until Success Rate				Visited Goals
			10%	20%	40%	100%	
RL2	3408.67	8.0 ± 1.0%	0	-	-	-	0.00 ± 0.0
ERL2	6811.58	22.0 ± 1.0%	2750	23750	-	-	1.49 ± 0.1
Max-Until-Exploit	**10374.25**	**43.7 ± 5.0%**	5000	15750	30750	-	**2.13 ± 0.2**
StDev-Until-Exploit	**11436.83**	**43.9 ± 7.5%**	0	14500	28000	-	**2.14 ± 0.3**
3-Reacher							
RL2	2500.25	26.0 ± 5.0%	7500	16250	-	-	0.87 ± 0.09
ERL2	3521.38	29.4 ± 10.5 %	0	11250	-	-	1.16 ± 0.12
Max-Until-Exploit	**9399.12**	**61.4 ± 2.0%**	1250	1750	4000	-	**1.70 ± 0.02**
StDev-Until-Exploit	**9626.29**	**58.9 ± 1.5%**	0	1500	4000	-	**1.70 ± 0.01**
3-DoomChoice							
RL2	3989.45	0.1375 ± 2.5%	0	8000	-	-	0.97 ± 0.02
ERL2	2076.17	0.1219 ± 2.8%	0	-	-	-	1.03 ± 0.06
Max-Until-Exploit	**5259.38**	**0.6687 ± 4.4%**	0	4500	7250	-	**2.02 ± 0.01**

Table 5.1: Results for the four environments. Area-under-Curve (AuC) integrates the learning curves for a measure of learning speed. Final Perf. is the final checkpoint performance. Updates until Success Rate is the number of gradient steps before a certain success rate is reached. Visited Goals is the average number of unique goals visited during exploration sub-episodes. 3-Doom-Choice results averaged over 2 seeds and 160 meta-episodes per data point. All other results averaged over 3 seeds and 1280 meta-episodes per data point.

presented in Tab. 5.1 and learning curves in Fig. 5.5. In-depth experimental details are described in Appendix A.

N-Monty-Hall: The "Monty-Hall" environment (see Fig. 5.3) is the class of MDPs described in Section 5.2. Each episode in the environment consists of an agent choosing from a set of $N+1$ actions. The first action is a NOOP, where the agent does not receive reward. The other N actions choose from a set of doors, behind which is an unobservable reward not known a-priori to the agent. For a given MDP from this class, only one fixed door will always give a small positive reward of $+0.1$ while all others give large negative rewards of -1. Succesful exploration in this environment therefore needs the agent to accumulate large negative reward in order to eventually find the positive goal. This setting clearly demonstrates RL2's risk-adverseness, and ERL2's difficulties due to reward sparsity.

N-Color-Choice: N-Color-Choice (see Fig. 5.4, top left) is a navigation environment, with N target absorbing goals $\{G_i\}_{i=1}^{N}$ randomly spawned within a grid. An agent starts within the grid facing a random orientation sampled from {North, East, West, South}. There are 3 possible actions the agent can take: move forward to a neighboring un-occupied grid cell along its orientation, or turn left or right to change orientation. Each goal G_i is assigned a hidden reward $r(G_i) \in \{-1, 1\}$ which is externally un-observable by the observation sensors within an episode, meaning there is no possibility to determine $r(G_i)$ using the observation history within a single episode. The rewards are set such that only one unique goal has positive reward $+1$.

N-Reacher: N-Reacher (Fig. 5.4, bottom left) is built off the benchmark environment "Reacher-v2" [182], extending that setting to multiple target positions and discrete torque actions of $\{-1, 0, 1\}$ N-Reacher has a 2-joint arm that must position its end-effector near a target position. The episode ends whenever the agent gets below a distance threshold to the target. There are N targets where only one of the targets has $+1$ and the others have -1. Each MDP in the class has target positions varying within the $[-1, 1] \times [-1, 1]$-sized area and random reward functions.

N-Doom-Choice: An implementation of the N-Color-Choice environment using the ViZ-Doom [216] engine. An agent starts in a room with a collection of N objects randomly scattered (Fig. 5.4, right). The agent must then learn which of the objects it should collect in order to gain a positive reward, with all other objects providing a negative reward if collected. The agent can only collect a single object in an episode. While similar to N-Color-Choice in semantics, this environment is much more challenging to learn, requiring longer time horizons meant to test the

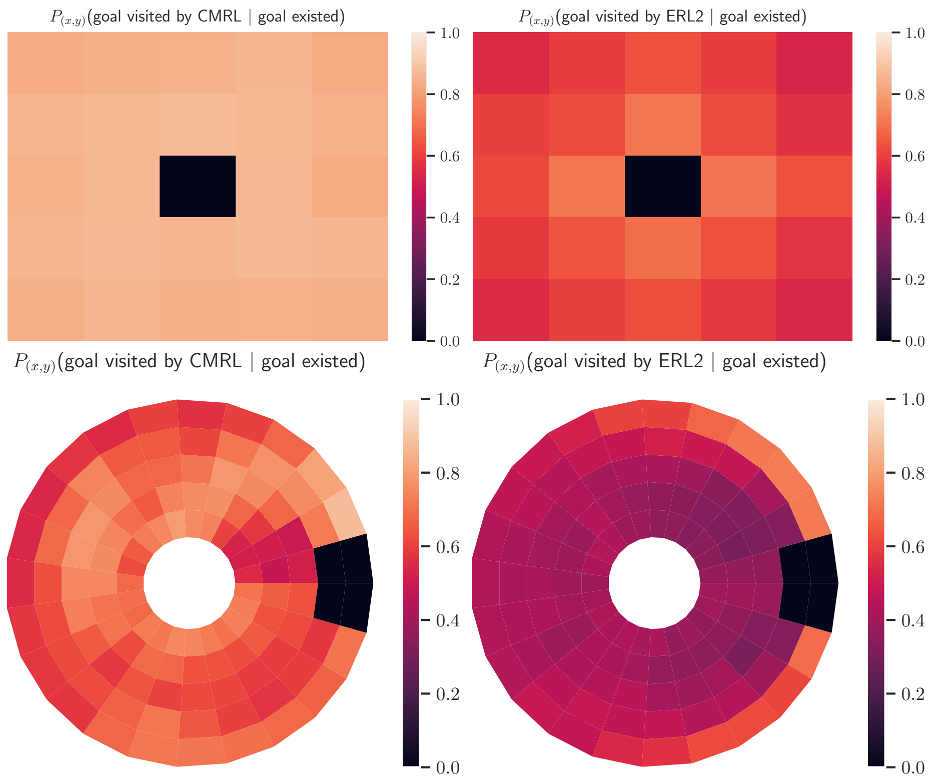


Figure 5.6: Exploration capability of CEMRL v.s. ERL2 in 3-Color-Choice (Top) and 3-Reacher (Bottom). Number of times CEMRL/ERL2 explored (x, y) position given that the goal was at (x, y), over 40000 random meta-episodes. Brighter is better. Continuous 3-Reacher goals are quantized in the visualizations. **Left:** CEMRL visitation frequency. **Right:** ERL2 visitation frequency.

scalability of the proposed CEMRL framework.

10-Monty-Hall Results: We can see a significant difference between CEMRL and the baselines in learning speed, final performance, and meta-learner exploration. As expected, RL2 fails at this environment, with the policy rapidly collapsing to the NOOP action as demonstrated by the learning curves in Fig. 5.5, converging to 0 success rate and 0 unique goals entered during exploration. In contrast, ERL2 performs better due to having no pressure to avoid entering the negative goals during explore sub-episodes, so it sustains an average of 6 goals entered. The difficulty is that it does not have incentive to explore, and the return of the exploit episode has to propagate backwards over a long time horizon to provide a signal to exploratory behaviour. The unique goals entered by ERL2 is slightly increasing, reaching 6.42 from an initial 6 after training for 12K update steps.

3-Color-Choice Results: For RL2, we can see a rapid collapse to no goal entries during exploration sub-episodes due to the high-risk involved. ERL2 is capable of learning but does so at a much slower pace than CEMRL. During exploration, ERL2 only learns to enter about half of the available goals. Despite the 1.5/3 average goals entered during exploration sub-episodes, ERL2 still has trouble making the connection between reward of goals entered and optimal exploit policy, only reaching average 0.2 exploit return, perhaps due to the longer time horizons making credit assignment more challenging. CEMRL learns faster and at a higher performance than baselines.

3-Reacher Results: CEMRL shows a substantial improvement in learning speed and final performance over RL2 and ERL2. RL2, in contrast to its performance in previous environments, actually demonstrated learning on this environment. The reward scales in 3-Color-Choice and 3-Reacher are high-risk but do not preclude RL2 from solving them like 10-Monty-Hall, only having a strong local minima for the NOOP policy which is then difficult to exit. For 3-Reacher, RL2 exited this local optimum after reaching it, as seen by the goal visitation curve dipping to 0 very early on in training and then recovering afterwards at a rate much higher than ERL2.

CEMRL Does Better Exploration: A measure of exploration optimality within the environments tested is the number of unique goals visited during exploration sub-episodes, which for N goals should be $> N$-2. We can see on the far-right column of Tab. 5.1 that only CEMRL achieves optimal exploration in 10-Monty-Hall (9/10), and 1.5-2x more unique goals in Color-Choice,

Model	AuC	Final Perf.	Updates until Success Rate					Visited Goals
			25%	50%	75%	95%	100%	
CEMRL								
Max-Until-Exploit	**9575.96**	**100 ± 0.0%**	1250	2750	3750	4500	5000	**9.00**
StDev-Until-Exploit	**9501.67**	99.7 ± 0.5%	1250	2500	3750	5000	6500	**9.00**
Mixing Schemes								
Max+StDev-Until-Exploit	9448.75	99.0 ± 0.8%	1250	2750	4000	5000	6500	**9.00**
No Reward Scheme								
Zero-Until-Exploit	9417.92	92.0 ± 11%	750	1750	3000	-	-	8.67
No Divergence								
Max-Until-Exploit	51.67	4.67 ± 6.6%	-	-	-	-	-	9.00
StDev-Until-Exploit	45.83	0.0 ± 0.0%	-	-	-	-	-	4.64
Zero-Until-Exploit	40.83	0.0 ± 0.0%	-	-	-	-	-	6.15

Table 5.2: CEMRL ablation results for 10-Monty-Hall. We examined the contribution of mixing reward schemes, using no reward scheme, and removing the divergence auxiliary objectives. We can see that mixing reward schemes gives around the same performance as either Max- or StDev-Until-Exploit. Using no reward scheme and divergence losses does much better than baselines in final performance and learning speed, but ends up with a lower final performance than the full CEMRL. Removing the divergence loss reduces learning capability in 10-Monty-Hall but interestingly Max-Until-Exploit still learns an optimal exploration policy.

Reacher and Doom-Choice.

State Coverage of CEMRL: To examine the variety of exploration behaviours of CEMRL and ERL2 after training in 3-Color-Choice and 3-Reacher, we sample 40000 meta-episodes for each environment and keep track of where the goals were sampled and whether the agent reached the goal, giving us an idea of the variety of states visited by the meta-learner during exploration. Fig. 5.6 shows this heatmap for both CEMRL and ERL2 in the left and right columns, respectively. CEMRL gets a much higher coverage of goal locations, meaning it not only improves over ERL2 on the number of unique goals visited during exploration sub-episodes, but the variety of such goals as well.

Ablation of CEMRL on 10-Monty-Hall: We carried out an ablation on 10-Monty-Hall to try to extract out the contribution of each component of our proposed CEMRL with results shown in Table 5.2. The first component we adjusted was the reward scheme. Given that the reward scheme adds an extra hyperparameter, we wanted to examine whether it was possible to mix together a variety of reward schemes and get similar results, to avoid an additional hyperparameter to tune. This was the row labeled "Max+StDev-Until-Exploit", a scheme which simply sums the Max- and StDev-Until-Exploit reward schemes together during exporation sub-episodes. The result demonstrates that we can nearly recover the performance of either reward scheme using this mixture, demonstrating the robustness of the hyperparameter and suggesting that it can reasonably be set to a mixture of reward schemes. The next component we changed was examining what happens when we remove all reward schemes, making CEMRL rely only on external exploit returns as ERL2 does. In this case, we can still see a substantial improvement over ERL2, which is (as explained later) mainly due to divergence auxiliary objective that encourages exploration. Despite the improvement over ERL2, Zero-Until-Exploit CEMRL still performs worse and with higher variance than if we had used a reward-sharing scheme. Finally, we examined what happens to CEMRL without the divergence auxiliary loss. In this case, we see a substantial decrease in exploit performance, with final performance near 0. Interestingly, the Max-Until-Exploit CEMRL eventually achieves an optimal explore policy despite the poor exploit performance. This suggests that higher exploit performance is likely achievable with a finer tuning of the entropy scale parameter used in A2C, to prevent the exploit sub-episode policy from collapsing "too early", i.e. before the explore sub-policy policies learn to enter goals. The other reward schemes do not seem successful at this challenging task without the divergence auxiliary loss.

Scaling up CEMRL: In order to test whether CEMRL can scale to more complex tasks, we report results on a 3D navigation environment [216]. From Fig. 5.5 and Tab. 5.1, we can see the clear advantage of the CEMRL Max-Until-Exploit model over the RL2 and ERL2 baselines. The baselines fail to learn above their initial random performance, perhaps owing to the increased timescales necessary in this more challenging environment (150 time steps total). These results provide evidence that the CEMRL framework, and the particular architectures presented here, can easily scale to more complex 3D navigation domains with high-dimensional image observations.

5.4 Discussion

Towards our goal of effective, general meta reinforcement learning agents using memory, in this chapter we presented an alternative concurrent framework for inter-episodic-memory-based meta reinforcement learning in general environments. Within this framework, interactions are processed in parallel using separate agents, with the meta-learning mechanism now driven by a stateful communication protocol between the separate rollout agents instead of an inter-episodic memory. Beyond the reduction of the meta horizon to the time horizon of the environment, the multi-agent setting also affords us the design and use of several specialized reward sharing schemes to induce risk-robustness, and divergence losses to encourage diverse agent behaviours. We demonstrated the effectiveness of our framework on a set of challenging environments, with concurrent meta architectures outperforming those in the sequential setting.

In the next chapter, we move on to describing a new inter-episodic memory architecture which is adapted from the highly-successful self-attention Transformer-XL architecture [40]. The proposed Gated Transformer-XL (GTrXL) is shown in comprehensive experiments to be highly effective in fully- and partially-observable meta-learning domains. Furthermore the GTrXL is shown to learn in a multitask meta-learning domain setting, where the same policy can learn state-of-the-art behaviour in a very disparate set of meta learning domains simultaneously and with the same model parameters.

Chapter 6

Transformers for Meta RL

It has been argued that self-attention architectures [207] deal better with longer temporal horizons than recurrent neural networks (RNNs): by construction, they avoid compressing the whole past into a fixed-size hidden state and they do not suffer from vanishing or exploding gradients in the same way as RNNs. Recent work has empirically validated these claims, demonstrating that self-attention architectures can provide significant gains in performance over the more traditional recurrent architectures such as the LSTM [40, 164, 45, 218]. In particular, the Transformer architecture [207] has had breakthrough success in a wide variety of domains: language modeling [40, 164, 218], machine translation [207, 51], summarization [126], question answering [42, 218], multi-task representation learning for NLP [45, 164, 218], and algorithmic tasks [42].

The repeated success of the transformer architecture in domains where sequential information processing is critical to performance makes it an ideal candidate for meta RL problems, where episodes can extend to thousands of steps and the critical observations for any decision often span the entire episode. Yet, the RL literature is dominated by the use of LSTMs as the main mechanism for providing memory to the agent [53, 102, 136]. Despite progress at designing more expressive memory architectures [73, 212, 177] that perform better than LSTMs in meta-learning and memory-based tasks and partially-observable environments, they have not seen widespread adoption in RL agents perhaps due to their complex implementation, with the LSTM being seen as the go-to solution for environments where memory is required. In contrast to these other memory architectures, the transformer is well-tested in many challenging domains and has seen several open-source implementations in a variety of frameworks [1].

Motivated by the transformer's superior performance over LSTMs and the widespread availability of implementations, in this chapter we investigate the transformer architecture in the RL setting. In particular, we find that the canonical transformer is significantly difficult to optimize, often resulting in performance comparable to a random policy. This difficulty in training transformers exists in the supervised case as well. Typically a complex learning rate schedule is required (e.g., linear warmup or cosine decay) in order to train [207, 40], or specialized weight initialization schemes are used to improve performance [164]. These measures do not seem to be sufficient for RL. In [134], for example, transformers could not solve even simple bandit tasks and tabular Markov Decision Processes (MDPs), leading the authors to hypothesize that the transformer architecture was not suitable for processing sequential information. Our experiments have also verified this observed critical instability in the original transformer model.

However in this chapter we succeed in stabilizing training with a reordering of the layer normalization coupled with the addition of a new gating mechanism to key points in the submodules of the transformer. Our novel gated architecture, the Gated Transformer-XL (GTrXL) (shown in Figure 6.1, Right), is able to learn much faster and more reliably and exhibit significantly better final performance than the canonical transformer. We further demonstrate that the GTrXL achieves state-of-the-art results when compared to the external memory architecture MERLIN [212] on the multitask meta-learning "DMLab-30" suite [10]. Additionally, we surpass LSTMs significantly on more partially-observable DMLab-30 levels while matching performance on the more reactive set of levels, as well as significantly outperforming LSTMs on memory-based continuous control and navigation environments. We perform extensive ablations on the GTrXL in challenging environments with both continuous actions and high-dimensional observations, testing the final performance of

[1] `www.github.com/tensorflow/tensor2tensor`

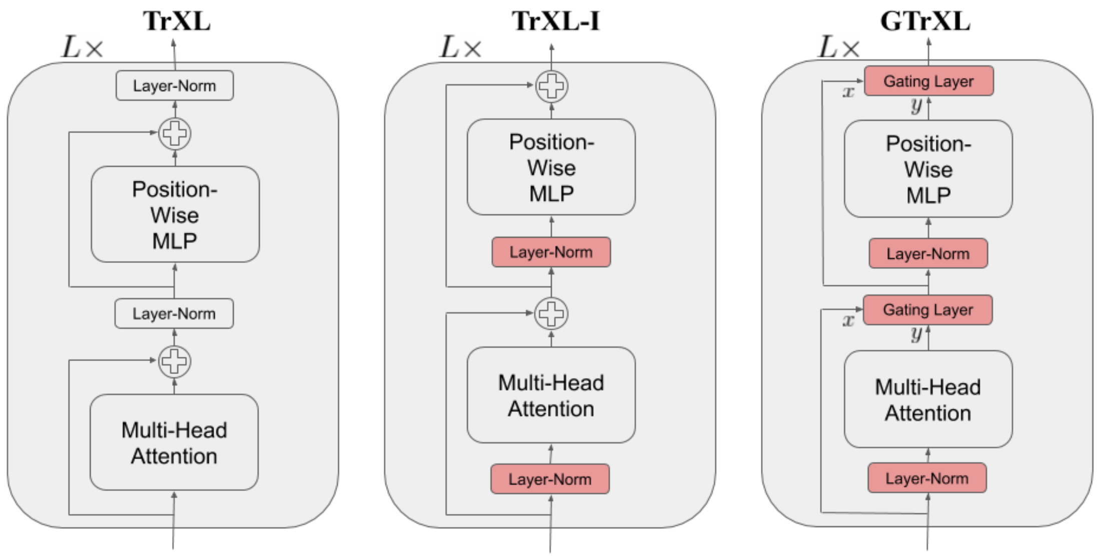


Figure 6.1: Transformer variants, showing just a single layer block (there are L layers total). **Left:** Canonical Transformer(-XL) block with multi-head attention and position-wise MLP submodules and the standard layer normalization [7] placement with respect to the residual connection [81]. **Center:** TrXL-I moves the layer normalization to the input stream of the submodules. Coupled with the residual connections, there is a gradient path that flows from output to input without any transformations. **Right:** The GTrXL block, which additionally adds a gating layer in place of the residual connection of the TrXL-I.

the various components as well as the GTrXL's robustness to seed and hyperparameter sensitivity compared to LSTMs and the canonical transformer. We demonstrate a consistent superior performance while matching the stability of LSTMs, providing evidence that the GTrXL architecture can function as a drop-in replacement to the LSTM networks ubiquitously used in RL and meta RL.

6.1 Background & Related Work

Gating has been shown to be effective to address the vanishing gradient problem and thus improve the learnability of recurrent models. LSTM networks [89, 72] rely on an input, forget and output gate that protect the update of the cell. GRU [31, 29] is another popular gated recurrent architecture that simplifies the LSTM cell, reducing the number of gates to two. Finding an optimal gating mechanism remains an active area of research, with other existing proposals [109, 99, 215], as well as works trying to discover optimal gating by neural architecture search [225] More generally, gating and multiplicative interactions have a long history [173]. Gating has been investigated previously for improving the representational power of feedforward and recurrent models [206, 41], as well as learnability [189, 224]. Initialization has played a crucial role in making deep models trainable [118, 69, 194].

There has been a wide variety of work looking at improving memory in reinforcement learning agents. External memory approaches typically have a regular feedforward or recurrent policy interact with a memory database through read and write operations. Priors are induced through the design of the specific read/write operations, such as those resembling a digital computer [212, 73] or an environment map [151, 76]. An alternative non-parametric perspective to memory stores an entire replay buffer of the agent's past observations, which is made available for the agent to itself reason over either through fixed rules [15] or an attention operation [162]. Others have looked at improving performance of LSTM agents by extending the architecture with stacked hierarchical connections / multiple temporal scales and auxiliary losses [94, 191] or allowing an inner-loop update to the RNN weights [132]. Other work has examined self-attention in the context of exploiting relational structure within the input-space [219] or within recurrent memories [177].

6.1.1 Transformer Architecture and Variants

The transformer network consists of several stacked blocks that repeatedly apply self-attention to the input sequence. The transformer layer block itself has remained relatively constant since its original introduction [207, 125, 164]. Each layer consists of two submodules: an attention operation followed by a position-wise multi-layer network (see Figure 6.1 (left)). The input to the transformer block is an embedding from the previous layer $E^{(l-1)} \in \mathbb{R}^{T \times D}$, where T is the number of time steps, D is the hidden dimension, and $l \in [0, L]$ is the layer index with L being the total number of layers. We assume $E^{(0)}$ is an arbitrarily-obtained input embedding of dimension $[T, D]$, e.g. a word embedding in the case of language modeling or a visual embedding of the per-timestep observations in an RL environment.

Multi-Head Attention

The Multi-Head Attention (MHA) submodule computes in parallel H soft-attention operations for every time step, producing an output tensor $Y^{(l)} \in \mathbb{R}^{T \times D}$. MHA operates by first calculating the query $Q^{(l)} \in \mathbb{R}^{H \times T \times d}$, keys $K^{(l)} \in \mathbb{R}^{H \times T \times d}$, and values $V^{(l)} \in \mathbb{R}^{H \times T \times d}$ (where $d = D/H$) through trainable linear projections $W_Q^{(l)}$, $W_K^{(l)}$, and $W_V^{(l)}$, respectively, and then using the combined Q, K, V, tensors to compute the soft attention. A residual connection [81] to the resulting embedding $E^{(l)}$ is then applied and finally layer normalization [7].

$\underline{\mathbf{MHA}(E^{(l-1)})}$:

$$\begin{aligned}
Q^{(l)}, K^{(l)}, V^{(l)} &= W_Q^{(l)} E^{(l-1)}, W_K^{(l)} E^{(l-1)}, W_V^{(l)} E^{(l-1)} \\
\alpha_{htm}^{(l)} &= Q_{htd} K_{hmd} \\
W_{htm}^{(l)} &= \text{MaskedSoftmax}(\alpha^{(l)}, \text{axis}{=}m) \\
\overline{Y}_{htd}^{(l)} &= W_{htm}^{(l)} V_{hmd}^{(l)} \\
\hat{Y}^{(l)} &= E^{(l-1)} + \text{Linear}(\overline{Y}^{(l)}) \\
Y^{(l)} &= \text{LN}(\hat{Y}^{(l)})
\end{aligned}$$

where we used Einstein summation notation to denote the tensor multiplications, MaskedSoftmax is a causally-masked softmax to prevent addressing future information, Linear is a linear layer applied per time-step and we omit reshaping operations for simplicity.

Multi-Layer Perceptron

The Multi-Layer Perceptron (MLP) submodule applies a 1×1 temporal convolutional network $f^{(l)}$ (i.e., kernel size 1, stride 1) over every step in the sequence, producing a new embedding tensor $E^{(l)} \in \mathbb{R}^{T \times D}$. As in [40], the network output does not include an activation function. After the MLP, there is a residual update followed by layer normalization:

$$E^{(l)} = \text{LN}(Y^{(l)} + f^{(l)}(Y^{(l)})) \tag{6.1}$$

Relative Position Encodings

The basic MHA operation does not take sequence order into account explicitly because it is permutation invariant, so positional encodings are a widely used solution in domains like language where order is an important semantic cue, appearing in the original transformer architecture [207]. To enable a much larger contextual horizon than would otherwise be possible, we use the relative position encodings and memory scheme described in [40]. In this setting, there is an additional $\mathcal{T}$-step memory tensor $M^{(l)} \in \mathbb{R}^{\mathcal{T} \times D}$, which is treated as constant during weight updates.

<u>**RMHA**$(M^{(l-1)}, E^{(l-1)})$:</u>

$$\begin{aligned}
\widetilde{E}^{(l-1)} &= [M^{(l-1)}, E^{(l-1)}] \\
Q^{(l)}, K^{(l)}, V^{(l)} &= W_Q^{(l)} E^{(l-1)}, W_K^{(l)} \widetilde{E}^{(l-1)}, W_V^{(l)} \widetilde{E}^{(l-1)} \\
R &= W_R^{(l)} \Phi \\
\alpha_{htm}^{(l)} &= Q_{htd} K_{hmd} + Q_{htd} R_{hmd} \\
&\quad + u_{h*d} K_{htm} + v_{h*d} R_{hmd} \\
W_{htm}^{(l)} &= \text{MaskedSoftmax}(\alpha^{(l)}, \text{axis}{=}m) \\
\overline{Y}_{htd}^{(l)} &= W_{htm}^{(l)} V_{hmd}^{(l)} \\
\hat{Y}^{(l)} &= E^{(l-1)} + \text{Linear}(\overline{Y}^{(l)}) \\
Y^{(l)} &= \text{LN}(\hat{Y}^{(l)})
\end{aligned}$$

where Φ is the standard sinusoid encoding matrix, $u^{(l)}, v^{(l)} \in \mathbb{R}^{H \times d}$ are trainable parameters, the $*$ represents the broadcast operation, and W_R is a linear projection used to produce the relative location-based keys (see [40] for a detailed derivation).

For shorthand, we write the relative position multi-head attention as:

$$Y^{(l)} = \text{LN}(E^{(l-1)} + \text{RMHA}(\text{SG}(M^{(l-1)}), E^{(l-1)})) \tag{6.2}$$

6.2 Gated Transformer Architectures

While the transformer architecture has achieved breakthrough results in modeling sequences for supervised learning tasks [207, 125, 40], a demonstration of the transformer as a useful RL memory has been notably absent. Previous work has highlighted training difficulties and poor performance [134]. When transformers have not been used for temporal memory but instead as a mechanism for attention over the input space, they have had success—notably in the challenging multi-agent Starcraft 2 environment [208]. Here, the transformer was applied solely across Starcraft units and not over time.

In order to alleviate these difficulties, we propose the introduction of powerful gating mechanisms in place of the residual connections within the transformer block, coupled with changes to the order of layer normalization in the submodules. Our gated architecture is motivated by multiplicative interactions having been successful at stabilizing learning across a wide variety of architectures [89, 189, 29], and we empirically see these same improvements in our proposed gated transformer.

Additionally, we propose that the key benefit of the “Identity Map Reordering”, where layer normalization is moved onto the "skip" stream of the residual connection, is that it enables an identity map from the input of the transformer at the first layer to the output of the transformer after the last layer. This is in contrast to the canonical transformer, where there are a series of layer normalization operations that non-linearly transform the state encoding. One hypothesis as to why the Identity Map Reordering improves results is, assuming that the submodules at initialization produce values that are in expectation near zero, the state encoding is passed un-transformed to the policy and value heads, enabling the agent to learn a Markovian policy at the start of training (i.e., the network is initialized such that $\pi(\cdot|s_t, \ldots, s_1) \approx \pi(\cdot|s_t)$ and $V^\pi(s_t|s_{t-1}, \ldots, s_1) \approx V^\pi(s_t|s_{t-1})$). The original positioning of the layer normalization, on the other hand, would scale down at each layer the information flowing through the skip connection, forcing the model to rely on the residual path. In many environments, reactive behaviours need to be learned before memory-based ones can be effectively utilized, i.e., an agent needs to learn how to walk before it can learn how to remember where it has walked.

6.2.1 Identity Map Reordering

Our first change is to place the layer normalization on only the input stream of the submodules, a modification described in several previous works [82, 164, 8]. The model using this *Identity Map Reordering* is termed TrXL-I in the following, and is depicted visually in Figure 6.1 (center). Because the layer norm reordering causes a path where two linear layers are applied in sequence, we apply a ReLU activation to each sub-module output before the residual connection. The TrXL-I already exhibits a large improvement in stability and performance over TrXL (see Section 6.4.3).

In more detail, the Identity Map Reordering modifies the standard transformer formulation as follows: the layer norm operations are applied only to the input of the sub-module and a non-linear ReLU activation is applied to the output stream.

$$\overline{Y}^{(l)} = \text{RMHA}(\text{LN}([\text{SG}(M^{(l-1)}), E^{(l-1)}])) \tag{6.3}$$

$$Y^{(l)} = E^{(l-1)} + \text{ReLU}(\overline{Y}^{(l)}) \tag{6.4}$$

$$\overline{E}^{(l)} = f^{(l)}(\text{LN}(Y^{(l)})) \tag{6.5}$$

$$E^{(l)} = Y^{(l)} + \text{ReLU}(\overline{E}^{(l)}) \tag{6.6}$$

See Figure 6.1 (Center) for a visual depiction of the TrXL-I.

6.2.2 Gating Layers

We further improve performance and optimization stability by replacing the residual connections in Equations 6.2 and 6.1 with gating layers. We call the gated architecture with the identity map reordering the *Gated Transformer(-XL)* (GTrXL). The final GTrXL layer block is written below:

$$\begin{aligned}
\overline{Y}^{(l)} &= \text{RMHA}(\ \text{LN}([\text{SG}(M^{(l-1)}), E^{(l-1)}])) \\
Y^{(l)} &= g^{(l)}_{\text{MHA}}(E^{(l-1)}, \text{ReLU}(\overline{Y}^{(l)})) \\
\overline{E}^{(l)} &= f^{(l)}(\text{LN}(Y^{(l)})) \\
E^{(l)} &= g^{(l)}_{\text{MLP}}(Y^{(l)}, \text{ReLU}(\overline{E}^{(l)}))
\end{aligned}$$

where g is a gating layer function. A visualization of our final architecture is shown in Figure 6.1 (right), with the modifications from the canonical transformer highlighted in red. In our experiments we ablate a variety of gating layers with increasing expressivity:

Input: The gated input connection has a sigmoid modulation on the input stream, similar to the short-cut-only gating from [82]:

$$g^{(l)}(x, y) = \sigma(W^{(l)}_g x) \odot x + y$$

Output: The gated output connection has a sigmoid modulation on the output stream:

$$g^{(l)}(x, y) = x + \sigma(W^{(l)}_g x - b^{(l)}_g) \odot y$$

Highway: The highway connection [189] modulates both streams with a sigmoid:

$$g^{(l)}(x, y) = \sigma(W^{(l)}_g x + b^{(l)}_g) \odot x + (1 - \sigma(W^{(l)}_g x + b^{(l)}_g)) \odot y$$

Sigmoid-Tanh: The sigmoid-tanh (SigTanh) gate [206] is similar to the Output gate but with an additional tanh activation on the output stream:

$$g^{(l)}(x, y) = x + \sigma(W^{(l)}_g y - b) \odot \tanh(U^{(l)}_g y)$$

Gated-Recurrent-Unit-type gating: The Gated Recurrent Unit (GRU) [31] is a recurrent network that performs similarly to an LSTM [89] but has fewer parameters. We adapt its powerful gating mechanism as an untied activation function in depth:

$$\begin{aligned}
r = \sigma(W^{(l)}_r y + U^{(l)}_r x), \qquad & z = \sigma(W^{(l)}_z y + U^{(l)}_z x - b^{(l)}_g) \\
\hat{h} &= \tanh(W^{(l)}_g y + U^{(l)}_g (r \odot x)) \\
g^{(l)}(x, y) &= (1 - z) \odot x + z \odot \hat{h}.
\end{aligned}$$

Gated Identity Initialization: We have claimed that the Identity Map Reordering aids policy optimization because it initializes the agent close to a Markovian policy / value function. If this is indeed the cause of improved stability, we can explicitly initialize the various gating mechanisms to be close to the identity map. This is the purpose of the bias $b^{(l)}_g$ in the applicable gating layers. We later demonstrate in an ablation that initially setting $b^{(l)}_g > 0$ can greatly improve learning speed.

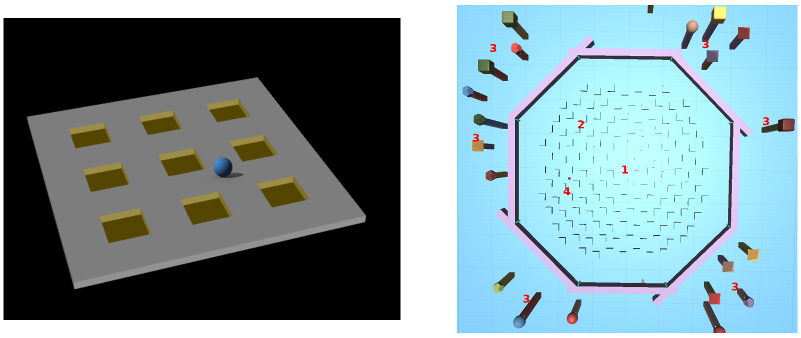


Figure 6.2: **Left:** The Numpad environment, showing the controllable "sphere" robot and a full 3x3 pad. Pads are activated when the robot collides with their center. The robot can move on the plane as well as jump to avoid pressing numbers. **Right:** Top down view of "Memory Maze": (1) Central chamber, (2) blocks among which the apple is placed, (3) landmarks the agent can use to locate the apple, (4) one of the possible location of the apple.

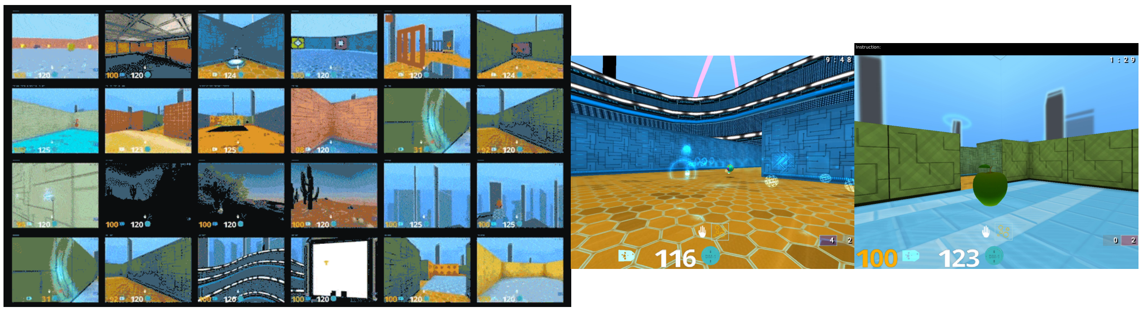


Figure 6.3: A set of example observations taken from DMLab-30 levels.

6.2.3 Motivating the Gated Transformer-XL Architecture

To provide a more detailed background on the two key architectural changes we have introduced into the canonical transformer, in this section we discuss the related work which served as the motivation behind these changes. The first change suggested is the movement of the layer normalization to the input of the residual module. The transformer canonically places the layer normalization after residual recombination, which causes an effective non-linearity on the path from input to output embedding. The move to change the layer normalization placement was primarily motivated by the results of GPT-2 at language modeling [164], which demonstrated effective learning using this same "pre-norm" placement, and also the ResNet-v2 architecture [82], which showed that maintaining an approximate identity map from input to output at initialization was important for improved performance and stability. Beyond the work done in this chapter, in concurrent work [217] the canonical "post-norm" placement was shown both theoretically and experimentally to cause optimization difficulties due to exploding gradients, and the same "pre-norm" placement as suggested by this chapter was shown to have improved stability and ease-of-optimization.

The gating layers added to the gated transformer were similarly motivated by the idea of improving optimization stability on the path from inputs to output embedding. In early experiments, we often observed high-magnitude gradients when training transformer policies with reinforcement learning objectives. High-magnitude gradients are a well-known problem in recurrent network design, where repeated summations can cause gradient magnitudes to explode [156] so, while distinct from the recurrent setting by notably lacking any recurrent weight matrix shared across depth, we were motivated to at least empirically test that recurrent gating layers [89][30] could provide an improvement to optimization stability and reduce the impact of, or even remove, high-magnitude gradients during training. Beyond improving recurrent optimization, previous work also demonstrated that gating layers help improve performance in completely feedforward architectures [41] [189] [206], which provided further motivation to empirically test this set of architectural changes to the transformer.

6.3 Methodology

6.3.1 Environment Details

Numpad Details

In Numpad [92], a robotic agent is situated on a platform resembling the 3x3 number pad of a telephone (generalizable to $N \times N$ pads). The agent can interact with the pads by colliding with them, causing them to be activated (visualized in the environment state as the number pad glowing). The goal of the agent is to activate a specific sequence of up to N^2 numbers, but without knowing this sequence a priori. The only feedback the agent gets is by activating numbers: if the pad is the next one in the sequence, the agent gains a reward of $+1$, otherwise all activated pads are cleared and the agent must restart the sequence. Each correct number in the sequence only provides reward once, i.e. each subsequent activation of that number will no longer provide rewards. Therefore the agent must explicitly develop a search strategy to determine the correct pad sequence. Once the agent completes the full sequence, all pads are reset and the agent gets a chance to repeat the sequence again for more reward. This means higher reward directly translates into how well the pad sequence has been memorized. An image of the scenario is provided in Figure 6.2. There is the restriction that contiguous pads in the sequence must be contiguous in space, i.e. the next pad in the sequence can only be in the Moore neighborhood of the previous pad. No pad can be pressed twice in the sequence.

Numpad (Fig. 6.2, left) has three actions, two of which move the sphere towards some direction in the x,y plane and the third allows the agent to jump in order to get over a pad faster. The observation consists of a variety of proprioceptive information (e.g. position, velocity, acceleration) as well as which pads in the sequence have been correctly activated (these will shut off if an incorrect pad is later hit), and the previous action and reward. Episodes last a fixed 500 steps and the agent can repeat the correct sequence any number of times to receive reward. Observations were processed using a simple 2-layer MLP with tanh activations to produce the transformer's input embedding.

DMLab-30 Details

DMLab-30 [10] (shown in Fig. 6.3) is a large-scale, multitask meta RL benchmark comprising 30 first-person 3D environments with image observations and has been widely used as a benchmark for architectural and algorithmic improvements [212, 53, 102, 86]. The levels test a wide range of agent competencies such as language comprehension, navigation, handling of partial observability, memory, planning, and other forms of long horizon reasoning, with episodes lasting over 4000 environment steps.

Example image observations are shown in Fig. 6.3. Ignoring the "jump" and "crouch" actions which we do not use, an action in the native DMLab action space consists of 5 integers whose meaning and allowed values are given in Table 6.1. Following previous work on DMLab [86], we used the reduced action set given in Table 6.2 with an action repeat of 4. Observations are 72×96 RGB images. Some levels require a language input, and for that all models use an additional 64-dimension LSTM to process the sentence.

Below is the complete list of DMLab-30 level names we use in our "Memory" and "Reactive" partitions for reproducibility.

Memory:

rooms_select_nonmatching_object, rooms_watermaze, explore_obstructed_goals_small, explore_goal_locations_small, explore_object_rewards_few, explore_obstructed_goals_large, explore_goal_locations_large, explore_object_rewards_many,

Reactive:

rooms_collect_good_objects_train, rooms_exploit_deferred_effects_train, rooms_keys_doors_puzzle, language_select_described_object, language_select_located_object, language_execute_random_task, language_answer_quantitative_question, lasertag_one_opponent_large, lasertag_three_opponents_large, lasertag_one_opponent_small, lasertag_three_opponents_small, natlab_fixed_large_map, natlab_varying_map_regrowth, natlab_varying_map_randomized, skymaze_irreversible_path_hard, skymaze_irreversible_path_varied, psychlab_arbitrary_visuomotor_mapping, psychlab_continuous_recognition, psychlab_sequential_comparison, psychlab_visual_search, explore_object_locations_small, explore_object_locations_large

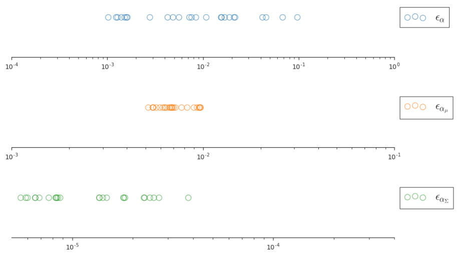


Figure 6.4: The 25 hyperparameter settings sampled for the sensitivity ablation (Sec. 6.4.3). X-axis is in log scale and values are sampled from the corresponding ranges given in Table 6.5.

Memory Maze Details

The "Memory Maze" environment (Fig.6.2) is a memory-based navigation task in which the agent must discover the location of an apple randomly placed in a maze. The agent receives a positive reward for collecting the apple and is then teleported to a random location in the maze, with the apple's position held fixed. The agent can make use of landmarks situated around the room to return as quickly as possible to the apple for subsequent rewards. Therefore, an effective mapping of the environment results in more frequent returns to the apple and higher reward.

An action in the native Memory Maze action space consists of 8 continuous actions and a single discrete action whose meaning and allowed values are given in Table 6.4. Unlike for DMLab, we used a hybrid continuous-discrete distribution [142] to directly output policies in the game's native action space. Observations are 72×96 RGB images. A visual description of the environment is shown in Fig. 6.2.

6.3.2 Agent Details

For all transformer architectures except when otherwise stated, we train relatively deep 12-layer networks with embedding size 256 and memory size 512. These networks are comparable to the state-of-the-art networks in use for small language modeling datasets (see enwik8 results in [40]). We chose to train deep networks in order to demonstrate that our results do not necessarily sacrifice complexity for stability, i.e. we are not making transformers stable for RL simply by making them shallow. Our networks have receptive fields that can potentially span any episode in the environments tested, with an upper bound on the receptive field of 6144 (12 layers $\times$ 512 memory [40]). Future work will look at scaling transformers in RL even further, e.g. towards the 52-layer network in [164].

For all experiments, we used V-MPO [188], an on-policy adaptation of Maximum a Posteriori Policy Optimization (MPO) [1, 2] that performs approximate policy iteration based on a learned state-value function $V(s)$ instead of the state-action value function used in MPO. Rather than directly updating the parameters in the direction of the policy gradient, V-MPO uses the estimated advantages to first construct a target distribution for the policy update subject to a sample-based KL constraint, then calculates the gradient that partially moves the parameters toward that target, again subject to a KL constraint. V-MPO was shown to achieve state-of-the-art results for LSTM-based agents on multi-task DMLab-30. As we want to focus on architectural improvements, we do not alter or tune the V-MPO algorithm differently than the settings originally presented in that paper for the LSTM architecture, which included hyperparameters sampled from a wide range.

Beyond sampling independent random seeds, each run also has V-MPO hyperparameters sampled from a distribution (see Table 6.5). The sampled hyperparameters are kept fixed across all models for a specific experiment, meaning that if one of the ϵ_α sampled is 0.002, then all models will have 1 run with $\epsilon_\alpha = 0.002$ and so on for the rest of the samples. The exception is for the DMLab-30 LSTM, where a more constrained range was found to perform better in preliminary experiments. Each model had 8 seeds started, but not all runs ran to completion due to compute issues. These hyperparameter settings were dropped randomly and not due to poor environment performance. We report how many seeds ran to completion for all models. At least 6 seeds finished for every model tested. We list architecture details by section below. All LSTM models have residual skip connections in depth.

All experiments in this work were carried out in an actor-learner framework [53] that utilizes TF-Replicator [20] for distributed training on TPUs in the 16-core configuration [70]. "Actors" running on CPUs performed network inference and interactions with the environment, and transmitted the resulting trajectories to the centralised “learner“.

Image Encoder

For DMLab-30 and Memory Maze, we used the image encoder architecture [188]. The ResNet was adapted from [86] and each of its layer blocks consists of a $(3 \times 3$, stride 1) convolution, followed by $(3 \times 3$, stride 2) max-pooling, followed by 2 3×3 residual blocks with ReLU non-linearities.

Agent Output

As in [188], in all cases we use a 256-unit MLP with a linear output to get the policy logits (for discrete actions), Gaussian distribution parameters (for continuous actions) or value function estimates.

Action name	Range
LOOK_LEFT_RIGHT_PIXELS_PER_FRAME	[-512, 512]
LOOK_DOWN_UP_PIXELS_PER_FRAME	[-512, 512]
STRAFE_LEFT_RIGHT	[-1, 1]
MOVE_BACK_FORWARD	[-1, 1]
FIRE	[0, 1]

Table 6.1: Native action space for DMLab. See `https://github.com/deepmind/lab/blob/master/docs/users/actions.md` for more details.

Action	Native DMLab action
Forward (FW)	`[ 0, 0, 0, 1, 0]`
Backward (BW)	`[ 0, 0, 0, -1, 0]`
Strafe left	`[ 0, 0, -1, 0, 0]`
Strafe right	`[ 0, 0, 1, 0, 0]`
Small look left (LL)	`[-10, 0, 0, 0, 0]`
Small look right (LR)	`[ 10, 0, 0, 0, 0]`
Large look left (LL)	`[-60, 0, 0, 0, 0]`
Large look right (LR)	`[ 60, 0, 0, 0, 0]`
Look down	`[ 0, 10, 0, 0, 0]`
Look up	`[ 0, -10, 0, 0, 0]`
FW + small LL	`[-10, 0, 0, 1, 0]`
FW + small LR	`[ 10, 0, 0, 1, 0]`
FW + large LL	`[-60, 0, 0, 1, 0]`
FW + large LR	`[ 60, 0, 0, 1, 0]`
Fire	`[ 0, 0, 0, 0, 1]`

Table 6.2: Simplified action set for DMLab from [86].

Action	Native DMLab action
Small look left (LL)	`[-10, 0, 0, 0, 0]`
Small look right (LR)	`[ 10, 0, 0, 0, 0]`
Look down	`[ 0, 10, 0, 0, 0]`
Look up	`[ 0, -10, 0, 0, 0]`
No-op	`[ 0, 0, 0, 0, 0]`

Table 6.3: Simplified action set for DMLab Arbitrary Visuomotor Mapping (AVM). This action set is the same as the one used for AVM in [212] but with an additional no-op, which may also be replaced with the *Fire* action.

Action name	Range
LOOK_LEFT_RIGHT	[-1.0, 1.0]
LOOK_DOWN_UP	[-1.0, 1.0]
STRAFE_LEFT_RIGHT	[-1.0, 1.0]
MOVE_BACK_FORWARD	[-1.0, 1.0]
HAND_ROTATE_AROUND_RIGHT	[-1.0, 1.0]
HAND_ROTATE_AROUND_UP	[-1.0, 1.0]
HAND_ROTATE_AROUND_FORWARD	[-1.0, 1.0]
HAND_PUSH_PULL	[-10.0, 10.0]
HAND_GRIP	{0, 1}

Table 6.4: Hybrid action set for Memory Maze, consisting of 8 continuous actions and a single discrete action.

Hyperparameter	Environment		
	DMLab-30	Numpad	Memory Maze
Batch Size	128	128	128
Unroll Length	95	95	95
Discount	0.99	0.99	0.99
Action Repeat	4	1	4
Pixel Control Cost	2×10^{-3}	-	-
Target Update Period	10	10	10
Initial η	1.0	10.0	1.0
Initial α	5.0	-	5.0
Initial α_μ	-	1.0	1.0
Initial α_Σ	-	1.0	1.0
ϵ_η	0.1	0.1	0.1
ϵ_α (log-uniform)	**LSTM** [0.001, 0.025) **TrXL** [0.001, 0.1)	-	[0.001, 0.1)
ϵ_{α_μ} (log-uniform)	-	[0.005, 0.01)	[0.005,0.01)
ϵ_{α_Σ} (log-uniform)	-	[5e-6, 4e-4)	[5e-6, 4e-5)

Table 6.5: V-MPO hyperparameters per environment.

Model	# Layers	Head Dim.	# Heads	Hidden Dim.	Memory Size	Runs Completed
LSTM	3	-	-	256	-	8
Large LSTM	12	-	-	512	-	6
TrXL	12	64	8	512	512	6
TrXL-I	12	64	8	512	512	6
GTrXL (GRU)	12	64	8	512	512	8
GTrXL (Input)	12	64	8	512	512	6
GTrXL (Output)	12	64	8	512	512	7
GTrXL (Highway)	12	64	8	512	512	7
GTrXL (SigTanh)	12	64	8	512	512	6
Thin GTrXL (GRU)	12	64	4	256	512	8

Table 6.6: DMLab-30 Ablation Architecture Details. We report the number of runs per model that ran to completion (i.e. 10 billion environment steps). We follow the standard convention that the hidden/embedding dimension of transformers is equal to the head dimension multiplied by the number of heads. (Sec. 6.4.1 & Sec. 6.4.3).

Model	# Layers	Head Dim.	# Heads	Hidden Dim.	Memory Size	Runs Completed
LSTM	3	-	-	256	-	5
GTrXL (GRU)	12	64	8	256	512	5

Table 6.7: Numpad Architecture Details. (Sec. 6.4.2).

Model	# Layers	Head Dim.	# Heads	Hidden Dim.	Memory Size
LSTM	3	-	-	256	-
TrXL	12	64	8	256	512
TrXL-I	12	64	8	256	512
GTrXL (GRU)	12	64	8	256	512
GTrXL (Output)	12	64	8	256	512

Table 6.8: Sensitivity ablation architecture details (Sec. 6.4.3).

Model	# Layers	Head Dim.	# Heads	Hidden Dim.	Memory Size	Runs Completed
GTrXL (GRU)	4	64	4	256	512	8

Table 6.9: Gated identity initialization ablation architecture details (Sec. 6.4.3).

Model	Mean HNR	Mean HNR, 100-capped
LSTM	99.3 ± 1.0	84.0 ± 0.4
TrXL	5.0 ± 0.2	5.0 ± 0.2
TrXL-I	107.0 ± 1.2	87.4 ± 0.3
MERLIN@100B	115.2	89.4
GTrXL (GRU)	117.6 ± 0.3	89.1 ± 0.2
GTrXL (Input)	51.2 ± 13.2	47.6 ± 12.1
GTrXL (Output)	112.8 ± 0.8	87.8 ± 0.3
GTrXL (Highway)	90.9 ± 12.9	75.2 ± 10.4
GTrXL (SigTanh)	101.0 ± 1.3	83.9 ± 0.7

Table 6.10: Final human-normalized return (HNR) averaged across all 30 DMLab levels for baselines and GTrXL variants. We also include the 100-capped score where the per-level mean score is clipped at 100, providing a metric that is proportional to the percentage of levels that the agent is superhuman. We see that the GTrXL (GRU) surpasses LSTM by a significant gap and exceeds the performance of MERLIN [212] trained for 100 billion environment steps. The GTrXL (Output) and the proposed reordered TrXL-I also surpass LSTM but perform slightly worse than MERLIN and are not as robust as GTrXL (GRU) (see Sec. 6.4.3). We sample 6-8 hyperparameters per model. We include standard error over runs.

6.4 Experiments

In this section, we provide experiments on a variety of challenging single and multi-task meta RL domains: DMLab-30 [10], Numpad and Memory Maze (see Fig. 6.2 and 6.3). These environments are meta-learning domains because they have randomized parameters each episode, and the solution to the current instance relies on discovering the latent parameters (what maze geometry the agent is within, where objects are located, what object it should recall in its past, etc.). Crucially we demonstrate that the proposed Gated Transformer-XL (GTrXL) not only shows substantial improvements over LSTMs on the partially-observable meta RL environments, but suffers no degradation of performance on more reactive environments. The GTrXL also exceeds MERLIN [212], an external memory architecture which used a Differentiable Neural Computer [73] coupled with auxiliary losses, surpassing its performance on both memory and reactive tasks.

6.4.1 Transformer as Effective RL Memory Architecture

We first present results of the best performing GTrXL variant, the GRU-type gating, against a competitive LSTM baseline, demonstrating a substantial improvement on the DMLab-30 domain [10]. Figure 6.5 shows mean return over all levels as training progresses, where the return is human normalized as done in previous work (meaning a human has a per-level mean score of 100 and a random policy has a score of 0), while Table 6.10 has the final performance at 10 billion environment steps. The GTrXL has a significant gap over a 3-layer LSTM baseline trained using the same V-MPO algorithm. Furthermore, we included the final results of a previously-published external memory architecture, MERLIN [212]. Because MERLIN was trained for 100 billion environment steps with a different algorithm, IMPALA [53], and also involved an auxiliary loss critical for the memory component to function, the learning curves are not directly comparable and we only report

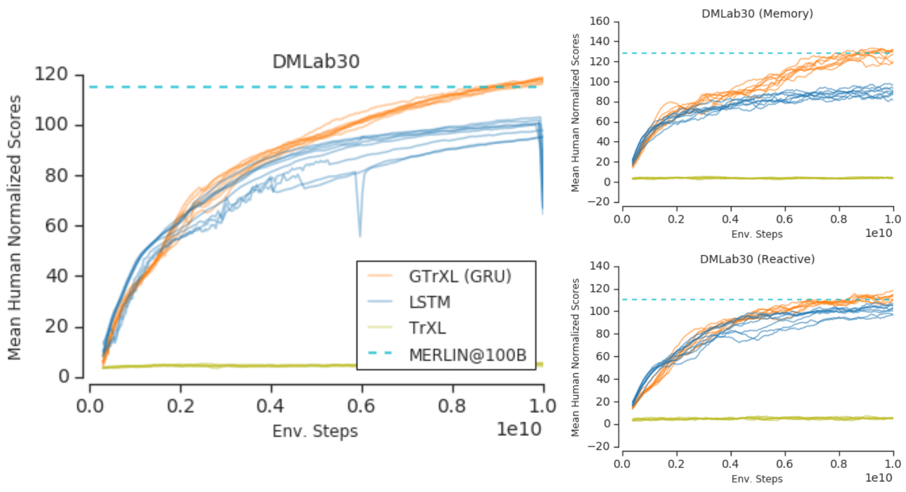


Figure 6.5: Average return on DMLab-30, re-scaled such that a human has mean 100 score on each level and a random policy has 0. **Left:** Results averaged over the full DMLab-30 suite. **Right:** DMLab-30 partitioned into a "Memory" and "Reactive" split (described in Table 6.3.1). The GTrXL has a substantial gain over LSTM in memory-based environments, while even slightly surpassing performance on the reactive set. We plot 6-8 hyperparameter settings per architecture. MERLIN scores obtained from personal communication with the authors.

the final performance of the architecture as a dotted line. Despite the differences, our results demonstrate that the GTrXL can match the previous state-of-the-art on DMLab-30. An informative split between a set of memory-based levels and more reactive ones (listed in Section 6.3.1) reveals that our model specifically has large improvements in environments where memory plays a critical role. Meanwhile, GTrXL also shows improvement over LSTMs on the set of reactive levels, as memory can still be used in some of these levels.

Furthermore, in [212], the DMLab Arbitrary Visuomotor Mapping task was specifically used to highlight the MERLIN architecture's ability to utilize memory. In Figure 6.6 we show that, given a similarly reduced action set as used in [212], see Table 6.3, the GTrXL architecture can also reliably attain human-level performance on this task.

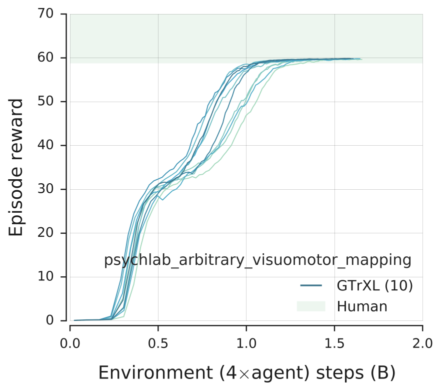


Figure 6.6: Learning curves for the DMLab Arbitrary Visuomotor Mapping task using a reduced action set.

6.4.2 Scaling with Memory Horizon

We next demonstrate that the GTrXL scales better compared to an LSTM when an environment's temporal horizon is increased, using the "Numpad" continuous meta RL task of [92] which allows an easy combinatorial increase in the temporal horizon. We present two results in this environment in Figure 6.9. The first measures the final performance of the trained models as a function of the pad size. We can see that LSTM performs badly on all 3 pad sizes, and performs worse as the pad size increases from 2 to 4. The GTrXL performs much better, and almost instantly solves the

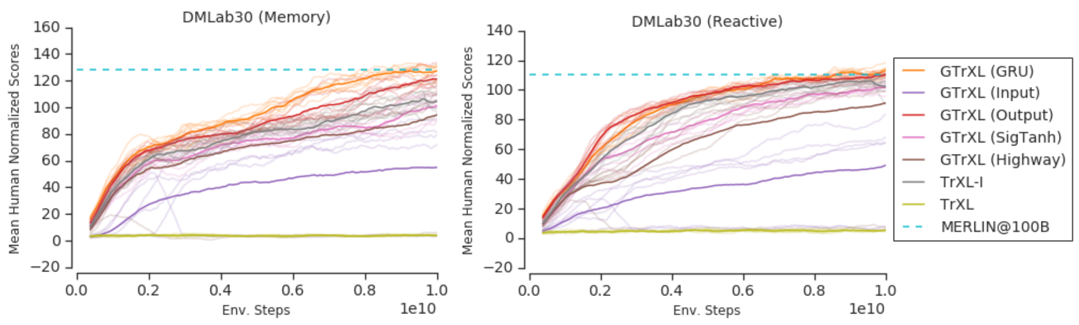


Figure 6.7: Learning curves for the gating mechanisms, along with MERLIN score at 100 billion frames as a reference point. We can see that the GRU performs as well as any other gating mechanism on the reactive set of tasks. On the memory environments, the GRU gating has a significant gain in learning speed and attains the highest final performance at the fastest rate. We plot both mean (bold) and the individual 6-8 hyperparameter samples per model (light).

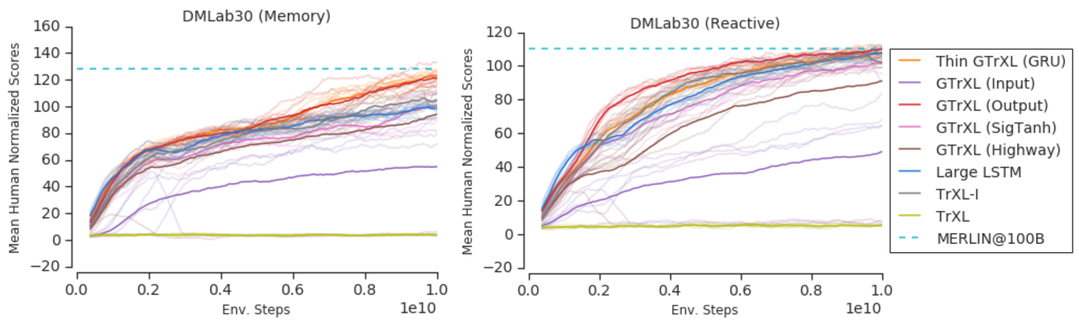


Figure 6.8: Learning curves comparing a thinner GTrXL (GRU) with half the embedding dimension of the other presented gated variants and TrXL baselines. The Thin GTrXL (GRU) has fewer parameters than any other model presented but still matches the performance of the best performing counterpart, the GTrXL (Output), which has over 10 million more parameters. We plot both mean (bold) and 6-8 hyperparameter settings (light) per model.

environment with its much more expressive memory. On the center and right images, we provide learning curves for the 2×2 and 4×4 Numpad environments, and show that even when the LSTM is trained twice as long it does not reach GTrXL's performance.

6.4.3 Gating Variants + Identity Map Reordering

We demonstrated that the GRU-type-gated GTrXL can achieve state-of-the-art results on DMLab-30, surpassing both a deep LSTM and an external memory architecture, and also that the GTrXL has a memory which scales better with the memory horizon of the environment. However, the question remains whether the expressive gating mechanisms of the GRU could be replaced by simpler alternatives. In this section, we perform extensive ablations on the gating variants described in Section 6.2.2, and show that the GTrXL (GRU) has improvements in learning speed, final performance and optimization stability over all other models, even when controlling for the number of parameters.

Performance Ablation

We first report the performance of the gating variants in DMLab-30. Table 6.10 and Figure 6.7 show the final performance and training curves of the various gating types in both the memory / reactive split, respectively. The canonical TrXL completely fails to learn, while the TrXL-I improves over the LSTM. Of the gating varieties, the GTrXL (Output) can recover a large amount of the performance of the GTrXL (GRU), especially in the reactive set, but as shown in Sec. 6.4.3 is generally far less stable. The GTrXL (Input) performs worse than even the TrXL-I, reinforcing the identity map path hypothesis. Finally, the GTrXL (Highway) and GTrXL (SigTanh) are more

Model	Mean Human Norm. Score	# Param. Millions
LSTM	99.3 ± 1.0	9.25M
Large LSTM	103.5 ± 0.9	51.3M
TrXL	5.0 ± 0.2	28.6M
TrXL-I	107.0 ± 1.2	28.6M
Thin GTrXL (GRU)	111.5 ± 0.6	22.4M
GTrXL (GRU)	117.6 ± 0.3	66.4M
GTrXL (Input)	51.2 ± 13.2	34.9M
GTrXL (Output)	112.8 ± 0.8	34.9M
GTrXL (Highway)	90.9 ± 12.9	34.9M
GTrXL (SigTanh)	101.0 ± 1.3	41.2M

Table 6.11: Parameter-controlled ablation. We report standard error of the means of 6-8 runs per model.

Model	% Diverged
LSTM	0%
TrXL	0%
TrXL-I	16%
GTrXL (GRU)	0%
GTrXL (Output)	12%

Table 6.12: Percentage of the 25 parameter settings where the training loss diverged within 2 billion env. steps. We do not report numbers for GTrXL gating types that were unstable in DMLab-30. For diverged runs we plot the returns in Figure 6.10 as 0 afterwards.

sensitive to the hyperparameter settings compared to the alternatives, with some settings doing worse than TrXL-I.

Hyperparameter and Seed Sensitivity

Beyond improved performance, we next demonstrate a significant reduction in hyperparameter and seed sensitivity for the GTrXL (GRU) compared to baselines and other GTrXL variants, using the "Memory Maze" environment (Fig.6.2) as a testbed. We chose to perform the sensitivity ablation on Memory Maze because (1) it requires the use of long-range memory to be effective and (2) it includes both continuous and discrete action sets which makes optimization more difficult. In Figure 6.10, we sample 25 independent V-MPO hyperparameter settings from a wide range of values and train the networks to 2 billion environment steps. Then, at various points in training (0.5B, 1.0B and 2.0B), we rank all runs by their mean return and plot this ranking. Models with curves which are both higher and flatter are thus more robust to hyperparameters and random seeds. Our results demonstrate that (1) the GTrXL (GRU) can learn this challenging memory environment in much fewer environment steps than LSTM, and (2) that GTrXL (GRU) beats the other gating variants in stability by a large margin, thereby offering a substantial reduction in necessary hyperparameter tuning. The values in Table 6.12 list what percentage of the 25 runs per model had losses that diverged to infinity. The only model reaching human performance in 2 billion environment steps is the GTrXL (GRU), with 10 runs having a mean score > 8.

Parameter-Count-Controlled Comparisons

For the final gating ablation, we compare transformer variants while tracking their total parameter count to control for the increase in capacity caused by the introduction of additional parameters in the gating mechanisms. To demonstrate that the advantages of the GTrXL (GRU) are not solely due to an increase in parameter count, we halve the number of attention heads (which also effectively halves the embedding dimension due to the convention that the embedding size is the number of heads multiplied by the attention head dimension). The effect is a substantial reduction in parameter count, resulting in less parameters than even the canonical TrXL. Fig. 6.8 and Tab. 6.11 compare the different models to the "Thin" GTrXL (GRU), with Tab. 6.11 listing the parameter counts. We include a parameter-matched LSTM model with 12 layers and 512 hidden size. The

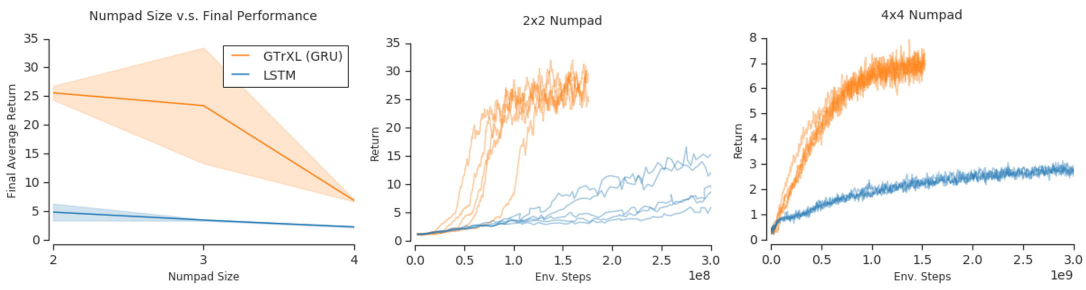


Figure 6.9: Numpad results demonstrating that the GTrXL has much better memory scaling properties than LSTM. **Left:** As the Numpad environment's memory requirement increases (because of larger pad size), the GTrXL suffers much less than LSTM. However, because of the combinatorial nature of Numpad, the GTrXL eventually also starts dropping in performance at 4x4. We plot mean and standard error of the last 200 episodes after training each model for 0.15B, 1.0B and 2.0B environment steps for Numpad size 2, 3 and 4, respectively. **Center, Right:** Learning curves for the GTrXL on 2×2 and 4×4 Numpad. Even when the LSTM is trained for twice as long, the GTrXL still has a substantial improvement over it. We plot 5 hyperparameter settings per model for learning curves.

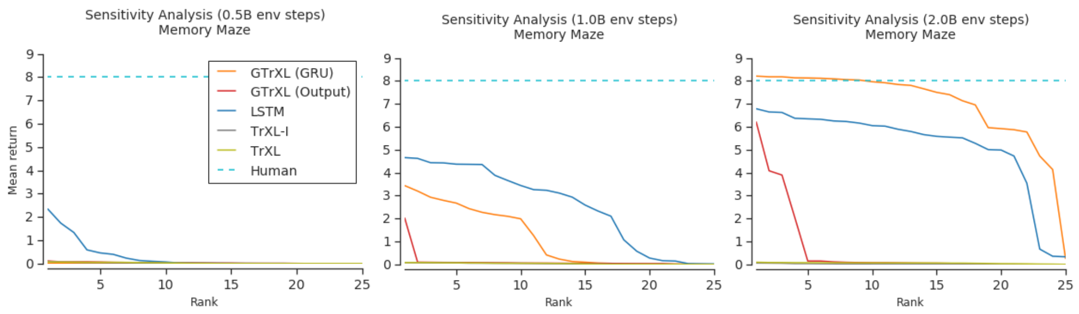


Figure 6.10: Sensitivity analysis of GTrXL variants versus TrXL and LSTM baselines. We sample 25 different hyperparameter sets and seeds and plot the ranked average return at 3 points during training (0.5B, 1.0B and 2.0B environment steps). Higher and flatter lines indicate more robust architectures. The same hyperparameter sampling distributions were used across models. We plot human performance as a dotted line.

Thin GTrXL (GRU) surpasses every other model except the GTrXL (GRU), even surpassing the next best-performing model, the GTrXL (Output), with over 10 million less parameters.

Gated Identity Initialization Ablation

All applicable gating variants in the main text were trained with the gated identity initialization. We observed in initial Memory Maze results that the gated identity initialization significantly improved optimization stability and learning speed. Figure 6.11 compares an otherwise identical 4-layer GTrXL (GRU) trained with and without the gated identity initialization. Similarly to the previous sensitivity plots, we plot the ranked mean return of 10 runs at various times during training. As can be seen from Fig. 6.11, there is a significant gap caused by the bias initialization, suggesting that preconditioning the transformer to be close to Markovian results in large learning speed gains.

6.4.4 Atari-57 Results

In this section, we run the GTrXL on the multitask Atari-57 benchmark (see Fig. 6.12 and Tab. 6.13). Although Atari-57 was not designed specifically to test an agent's memory capabilities, we include these results here to demonstrate that we suffer no performance regression on a popular environment suite, providing further evidence that GTrXL can be used as an architectural replacement to the LSTM.

The LSTM and GTrXL are matched in width at 256 dimensions. The GTrXL is 12 layers deep to show our model's learning stability even at large capacity. The LSTM architecture matches the

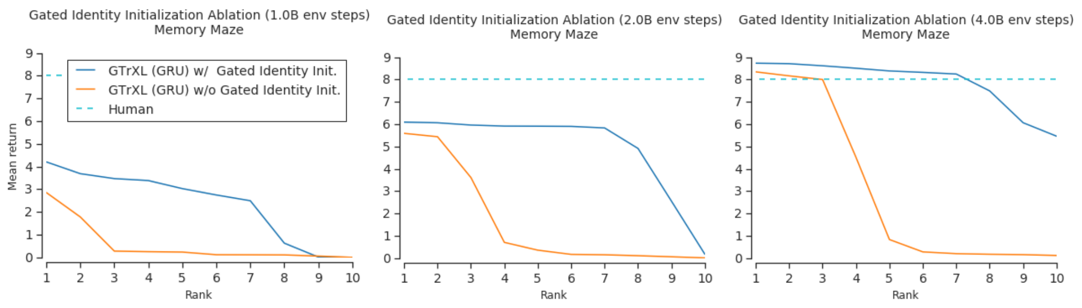


Figure 6.11: Ablation of the gated identity initialization on Memory Maze by comparing 10 runs of a model run with the bias initialization and 10 runs of a model without. Every run has independently sampled hyperparameters from a distribution. We plot the ranked mean return of the 10 runs of each model at 1, 2, and 4 billion environment steps. Each mean return is the average of the past 200 episodes at the point of the model snapshot. We plot human performance as a dotted line.

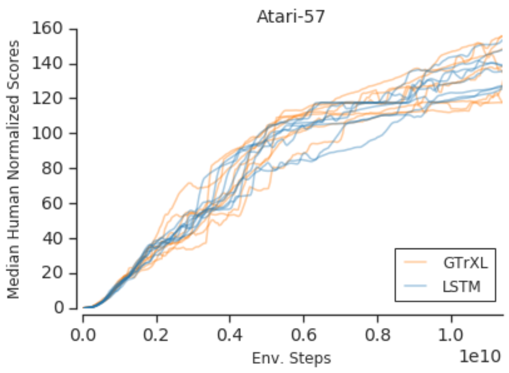


Figure 6.12: Median human-normalized returns as training progresses for both GTrXL and LSTM models. We run 8 hyperparameter settings per model.

one reported in [188]. We train for 11.4 billion environment steps, equivalent to 200 million per environment. We run 8 hyperparameter settings per model.

6.4.5 Motivating the Identity Map Re-ordering

In this section, we present further evidence corroborating the hypothesis that the identity map re-odering of the layer normalization operation can enable an initial reactive policy compared to other placements of the layer norm.

To show this, we sampled a randomly-initialized deep residual network. For simplicity, we used a single fully-connected layer in each residual stream in place of the MLP or MHA sub-module in the transformer blocks. Each layer is randomly initialized using the PyTorch [157] "torch.nn.Linear" default initializer and is not using an activation function. Let x be the input to a residual block, $f(x)$ the residual module, and y the output of the residual block. We consider 4 different placements of layer normalization within these randomly-initialized networks.

1. **nonorm** ($y = x + f(x)$): represents a standard residual network without any normalization operations.

Model	Median HNR
LSTM	136.6 $\pm$ 3.4
GTrXL	137.1 $\pm$ 5.0

Table 6.13: Final human-normalized median return across all 57 Atari levels for LSTM and GTrXL at 11.4 billion environment steps (equivalent to 200 million per individual game). Both models are 256 dimensions in width. We include standard error over runs.

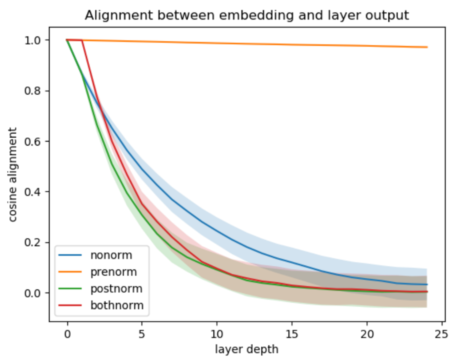


Figure 6.13: Alignment between embedding (network input) and layer features for a randomly-initialized deep linear residual network. We can see that the only layer norm placement which maintains high alignment is the identity map re-ordering (prenorm for short in the legend). Averaged over 1000 random 256-dimensional embeddings sampled from a standard Gaussian. Shaded regions represents standard error.

2. **prenorm** ($y = x + f(LN(x))$): is shorthand for the identity map re-ordering, where layer normalization is applied only before input to the residual stream fully-connected layer.
3. **postnorm** ($y = LN(x + f(x))$): is the original transformer layer norm placement, applied after residual recombination.
4. **bothnorm** ($y = LN(x + f(LN(x)))$): combines prenorm and postnorm, i.e. applies 2 layer norms per residual block: one applied on the input to the residual stream, and the other applied after residual recombination.

We design an experiment to measure how easily an input state embedding will pass unchanged to the policy output layer for each layer norm placement described above. To do this, we sample 1000 256-dimensional embeddings from a standard Gaussian and pass it through a 24-layer linear residual network, constructed as described earlier. We then evaluate the cosine alignment of the input embedding to each layer's feature embedding, to see how much noise is injected as the embedding progresses through the residual blocks. We show in Fig. 6.13 that the only layer norm placement that maintains high alignment to the input embedding as a function of depth is the prenorm placement, or identity map re-ordering. In particular, we can see that the postnorm, bothnorm and to a lesser extent nonorm placements exhibit a very rapid decay of initial feature alignment, meaning by the 12th layer extracting information directly from the state becomes much more difficult. In contrast, prenorm is almost always aligned to the initial embedding, with barely any degradation throughout the entire 24-layers.

6.5 Discussion

To summarize, in this chapter we provided evidence that confirms previous observations in the literature that standard transformer models are unstable to train in the RL setting and often fail to learn completely [134]. We presented a new architectural variant of the transformer model, the GTrXL, which has increased performance, more stable optimization, and greater robustness to initial seed and hyperparameters than the canonical architecture. The key contributions of the GTrXL are reordered layer normalization modules and a gating layer instead of the standard residual connection. We performed extensive ablation experiments in meta RL domains testing the robustness, ease of optimization and final performance of the gating layer variations, as well as the effect of the reordered layer normalization. These results empirically demonstrate that the GRU-type gating performs best across all metrics, exhibiting comparable robustness to hyperparameters and random seeds as an LSTM while still maintaining a performance improvement. Furthermore, the GTrXL (GRU) learns faster, more stably and achieves higher final performance (even when controlled for parameters) than the other gating variants on the challenging multitask DMLab-30 benchmark suite. Having demonstrated substantial and consistent improvement in DMLab-30, Numpad and

Memory Maze over the ubiquitous LSTM architectures currently in use, the GTrXL makes the case for wider adoption of transformers in RL and meta learning.

However, a major obstacle not directly addressed in this chapter is the significant computational resources required to train even moderate-scale transformer models. While also an issue in training transformers using standard supervised learning, this cost is compounded when training using (non-offline) reinforcement learning due to the fact that the learning policy (or a policy close to it) must be used for acting. With a forward pass of the transformer thus necessary for each environment step taken during training, substantial computational resources are required to produce enough data in a reasonable amount of time. In this chapter, we sidestepped this obstacle through running a large number of distributed CPU actors in parallel; however, these resources are not typically available except at large industry labs or when using university supercomputers. Therefore, in order to democratize the use of transformers in (meta-)reinforcement learning and unlock their potential for more researchers, in the next chapter we work towards one solution to efficiently training transformers with reinforcement learning, Actor-Learner Distillation.

Chapter 7

Efficient Transformers for Meta RL

In Chapter 6, we demonstrated that large-scale transformers were capable of reaching new state-of-the-art performance on challenging multi-task meta-learning benchmarks. However, many real-world applications such as robotics provide hard constraints on power and compute that limit the viable model complexity of agents. Similarly, in many distributed RL settings, acting is done on un-accelerated hardware such as CPUs, which likewise restricts model size to prevent intractable experiment run times or the requirement of substantial computational resources, as was the case of Chapter 6. These "actor-latency" constrained settings present a major obstruction to the further scaling up of model complexity that has recently been extremely successful in supervised learning [19].

To be able to utilize large model capacity while still operating within the limits imposed by the system during acting, in this chapter we develop an "Actor-Learner Distillation" (ALD) procedure that leverages a continual form of distillation that transfers learning progress from a large capacity learner model to a small capacity actor model. We then apply this ALD procedure to significantly reduce the computational cost of training gated transformers in partially-observed meta-reinforcement learning domains. With gated transformer models as the learner and LSTMs as the actor, we demonstrate in challenging memory and meta-learning environments that using Actor-Learner Distillation recovers the clear sample-efficiency gains of the gated transformer learner model, while still maintaining the fast inference and reduced total training time of the LSTM actor model.

Compared to standard supervised learning domains, reinforcement learning presents unique challenges in that the agent must act while it is learning. In certain *actor-latency-constrained* application areas, there exists maximum latency constraints on the acting policy which limit its model size. These constraints on latency preclude typical solutions to reducing the computational cost of high capacity models, such as model compression or off-policy reinforcement learning, as it strictly requires a low computational-complexity model to be acting during learning. Here, the major constraint is that the acting policy model must execute a single inference step within a fixed budget of time, which we denote by T_{actor} – the amount of compute or resources used during learning is in contrast not highly constrained. This setting is ubiquitous within real-world application areas: for example, in the context of learning policies for robotic platforms, because of inherent limitations in compute ability due to power and weight considerations it is unlikely a large model could run fast enough to provide actions at the control frequency of the robot's motors.

However, for many of these strictly actor-latency constrained settings there are orthogonal challenges involved which prevent ease of experimentation, such as the requirement to own and maintenance real robot hardware. In order to develop a solution to actor-latency constrained settings without needing to deal with substantial externalities, we focus on the related area of distributed on-policy reinforcement learning [136, 182, 54]. Here a central learner process receives data from a series of parallel actor processes interacting with the environment. The actor processes run step-wise policy inference to collect trajectories of interaction to provide to the learner, and they can be situated adjacent to the accelerator or distributed on different machines. With a large model capacity, the bottleneck in experiment run-times for distributed learning quickly becomes actor inference, as actors are commonly run on CPUs or devices without significant hardware acceleration available. The simplest solution to this constraint is to increase the number of parallel actors, often resulting in excessive CPU resource usage and limiting the total number of experiments that can be run on a compute cluster. Therefore, while not a hard constraint, experiment run-time in this setting is largely dominated by actor inference speed. The distributed RL setting therefore presents itself as a accessible test-bed for solutions to actor-latency constrained reinforcement learning.

Within the domain of distributed RL, an area where reduced actor-latency during learning could make a significant impact is in the use of large Transformers [207] to solve partially-observable environments [155]. Transformers [207] have rapidly emerged as the state-of-the-art architecture across a wide variety of sequence modeling tasks [19, 164, 45] owing to their ability to arbitrarily and instantly access information across time as well as their superior scaling properties compared to recurrent architectures. As shown in Chapter 6, their application to reinforcement learning domains has shown results surpassing previous state-of-the-art architectures while matching standard LSTMs in robustness to hyperparameter settings [155]. However, a downside to the Transformer compared to LSTM models is its significant computational cost.

In this chapter, we present a solution to actor-latency constrained settings, "Actor-Learner Distillation" (ALD), which leverages a continual form of Policy Distillation [174, 149] to compress, online, a larger "learner model" towards a tractable "actor model". In particular, we focus on the distributed RL setting applied to partially-observable and meta-reinforcement-learning environments, where we aim to be able to exploit the transformer model's superior *sample-efficiency* while still having parity with the LSTM model's *computational-efficiency* during acting. On challenging memory environments where the transformer has a clear advantage over the LSTM, we demonstrate our Actor-Learner Distillation procedure provides substantially improved sample efficiency while still having experiment run-time comparable to the smaller LSTM.

7.1 Background & Related Work

A Markov Decision Process (MDP) [196] is a tuple of $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \gamma, \mathcal{R})$ where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite action space, $\mathcal{T}(s'|s, a)$ is the transition model, $\gamma \in [0, 1]$ is a discount factor and $\mathcal{R}$ is the reward function. A stochastic policy $\pi \in \Pi$ is a mapping from states to a probability distribution over actions. The value function $V^{\pi}(s)$ of a policy π for a particular state s is defined as the expected future discounted return of starting at state s and executing π: $V^{\pi}(s) = \sum_{t=0}^{\infty} \gamma^t r_t$. The optimal policy π^* is defined as the policy with the maximum value at every state, i.e. $\forall s \in \mathcal{S}, \pi \in \Pi$ we have $V^{\pi^*}(s) > V^{\pi}(s)$. The optimal policy is guaranteed to exist [196]. In this chapter's experiments, we focus on Partially-Observable MDPs (POMDPs) which define environments which cannot observe the state directly and instead must reason over observations. POMDPs require the agent to reason over histories of observations in order to make an informed decision over which action to choose. This is what motivates our use of memory models such as LSTMs [89] or gated transformers (Chapter 6).

In this chapter, similar to the previous one, we use V-MPO [188] as the main reinforcement learning algorithm. V-MPO, an on-policy value-based extension of the Maximum a Posteriori Policy Optimisation algorithm [3], used an EM-style policy optimization update along with regularizing KL constraints to obtain state-of-the-art results across a wide variety of environments. Similar to [188], we use the IMPALA [54] distributed RL framework to parallelize acting and learning. Actor processes step through the environment and collect trajectories of data to send to the learner process, which batches actor trajectories and optimizes reinforcement learning objectives to update the policy parameters. The updated parameters are then communicated back to actor processes in order to maintain on-policyness.

In the following, we first refer to a single observation and associated statistics (reward, etc.) as a **step**. We refer to an **environment step** as the acquisition of a step from the environment after an action is taken. The total number of environment steps is a measure of the sample complexity of an RL algorithm. We refer to a step processed by an RL algorithm as an **agent step**. The total number of agent steps is a measure of the computational complexity of the RL algorithm. We use **steps per second** (**SPS**) to measure the speed of an algorithm. We refer to **Learner SPS** as the total number of agent steps processed by the learning algorithm per second. We refer to **Actor SPS** as the total number of environment steps acquired by a single actor per second.

Related to this chapter's presented Actor-Learner Distillation (ALD) method, whether a deep neural network could be compressed into a simpler form was examined shortly after deep networks were shown to have widespread success [6, 88]. Distillation objectives revealed that not only could high-capacity teacher networks be transferred to low-capacity student networks [6], but that training using these imitation objectives produced a superior low-capacity model compared to training from scratch [88], with this effect generalizing to the case where student and teacher share equivalent capacity [66]. Similar to this chapter's desired goal to leverage the rapid memorization capabilities of transformers while training an LSTM, other work showed that distillation between different architecture classes could enable the transfer of inductive biases [112, 4]. Within RL, distillation has most often been used as a method for stabilizing multitask learning [174, 149, 201, 12].

The method most closely related to ALD in the literature is Mix&Match [38], where an ensemble of policies is defined such that each successive model in the ensemble has increasing capacity. All models in the ensemble are distilled between each other, and a mixture of the ensemble is used for sampling trajectories, with the mixture coefficients learnt through population-based training [95]. There are however significant difference from ALD: the distillation procedure was meant to transfer simple, quick-learning skills from a low-capacity model to a high-capacity model (the opposite intended direction of the benefit of distillation in ALD) and a mixture policy was used to sample trajectories which did not consider actor-latency constraints (and preferred sampling from the larger model if it performed better).

Similar to ALD, various works have looked at exploiting asymmetries between acting and learning in reinforcement learning [159, 93]. In [159], since the value function is ultimately not needed during deployment, it was given privileged access to state information during training (whereas the policy must operate directly from pixels). Our baseline Asymm. AC was derived from this insight that value networks are solely needed during learning, but our variant had a value function that utilized privileged compute resources instead of state access during training. Another closely related method is the training paradigm of [93], where a belief network was trained to predict hidden state information in partially-observable meta-learning environments. The hidden representations from the belief network were then used as an auxiliary input to an agent network trained to solve the task, enabling representation learning to use privileged information during training.

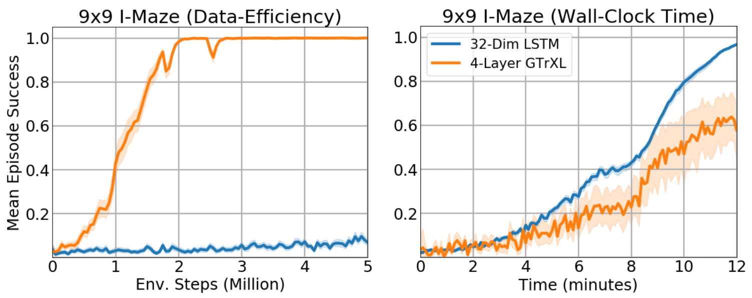
As an alternative to model compression, architecture development can reduce the computational cost of certain model classes. Transformers in particular have had a large amount of attention towards designing more efficient alternatives [200], as gains from increased model scale have yet to saturate [19]. This architectural development has taken many forms: sparsified attention [28], compressed attention [166, 39], use of kNN instead of soft-attention [115], etc. (See [200] for a comprehensive review). However, architectural development is largely orthogonal to the ALD procedure, as it means we can further scale actor and learner models by a corresponding degree.

7.2 Actor-Learner Distillation

Within the setting of distributed RL, a major challenge in applying transformers to reinforcement learning is their significant computational cost owing to their high actor latency. As an example, in Fig. 7.1 we present some results comparing a transformer to an LSTM on the I-Maze memory environment (see Sec. 7.5.1). In Fig. 7.1, Left, a 4-layer Gated Transformer-XL (GTrXL) [155] agent significantly surpasses a 32-dim. LSTM in terms of data efficiency, with the x-axis as environment steps. However, in the right diagram, when the x-axis is switched to wall-clock time the LSTM becomes the clearly more efficient model.

In order to work towards an ideal model with both high sample efficiency and low experiment run time, we analyzed which parts of the distributed actor-critic system led to the transformer's main computational bottlenecks. As shown in the table on the bottom of Fig. 7.1, we found the percentage of time that the asynchronous learner process was spending waiting for new trajectory data was substantially higher for the transformer than for the LSTM. Further looking at actor SPS between models revealed that the cause of this idling is mainly due to the transformer's actor SPS being over 15 times slower than the LSTM. This substantial decrease in SPS could even render the transformer not viable in actor-latency-constrained settings, e.g. robotic control environments, which entails a hard constraint on actor inference speed.

These findings suggest that considerable improvements in training speed could be attainable if the actors produced data faster. While de-coupling actor and learner speeds using an off-policy RL algorithm could be a suitable solution for many applications, we desired a solution that respected strict constraints on actor latency, which precludes ever running a large model during inference. As an alternative solution which respected this constraint, we designed a model compression procedure termed "Actor-Learner Distillation" (ALD) which continually performs distillation between a large-capacity "Learner" model, which is trained using RL but never run directly in the environment, and



Model	% Idle for Traj.	Actor SPS
LSTM	34%	1053
GTrXL	84%	70

Figure 7.1: **Left:** On the I-Maze environment (see Sec. 7.5.1), there is a clear sample efficiency advantage of the Gated Transformer-XL (GTrXL) [155], which rapidly reaches 100% success rate compared to the LSTM. **Center:** When the x-axis is changed to wall-clock time, the LSTM becomes the more efficient model. **Right:** Table showing, for both 32-dim. LSTM and 4-layer GTrXL, (1) percentage of time the learner spends waiting for new on-policy trajectory data from distributed actor processes, (2) how many environment steps are processed per second (SPS) on a single actor process. We can see that for the much more computationally expensive transformer, the learner is spending a majority of its time idling and this is due to the order-of-magnitudes slower actor inference. Plots averaged over 3 seeds, all run using reference machine A (Appendix 7.4.3).

a fast "Actor" model, which is trained using distillation from the learner and is used to collect data.

As we use an actor-critic training paradigm in this work, each of these models effectively comprises two functions, the policy and value function. We denote the actor model by $\mathcal{M}_A = (\pi_A, V^\pi_A)$ and learner model by $\mathcal{M}_L = (\pi_L, V^\pi_L)$. The actor model is chosen with the appropriate model capacity to compute an inference step on the target hardware as fast as possible, i.e. within time T_{actor}. In contrast, there are far fewer constraints on learner model $\mathcal{M}_L$, as it is typically running on a central process with accelerated hardware. While there are no restrictions besides latency concerns on the model class of actor and learner in the context of ALD, in this work we choose to focus on the case of the actor model being an LSTM and the learner model being a transformer. As there is a clear data efficiency advantage to the transformer and a clear computational advantage to the LSTM, the success of ALD at extracting the benefits of both models can be more clearly recognized.

Actor-Learner Distillation proceeds with the learner model $\mathcal{M}_L$ being trained using a standard reinforcement learning algorithm, with the exception of the environment data being generated by the actor model. The actor model is trained using a policy distillation loss:

$$L^\pi_{ALD} = \mathbb{E}_{s\sim\pi_A}\left[\mathcal{D}_{KL}(\pi_A(\cdot|s)||\pi_L(\cdot|s))\right] = \mathbb{E}_{s\sim\pi_A}\left[\sum_{a\in\mathcal{A}} \pi_A(a|s)\log\frac{\pi_A(a|s)}{\pi_L(a|s)}\right] \tag{7.1}$$

Similar to previous distillation work in the multitask setting [201], we employ this loss bidirectionally: the actor is trained towards the learner policy and the learner policy is regularized towards the actor policy. This regularization loss on the learner was seen to enable smoother optimization. As actor and learner policy will naturally be different at some points during training, an off-policy RL algorithm for the learner could be thought to be required if the divergence is large enough. We found leveraging off-policy data to be effective for the actor, and sampled trajectory data from a replay buffer when computing Eq. 7.1 during actor model optimization.

Beyond a distillation loss, we experimented with a value distillation loss:

$$L^V_{ALD} = \mathbb{E}_{s\sim\pi_A}\left[\frac{1}{2}(V^\pi_L(s) - V^\pi_A(s))^2\right] \tag{7.2}$$

This loss functions to encourage actor model representations to model task reward structure. The use of value distillation in order to improve representations in a policy network has also been explored in concurrent work [35]. Unlike the policy distillation loss, this loss is only used to train the actor model. The final Actor-Learner Distillation loss is:

$$L_{ALD} = \alpha_\pi L^\pi_{ALD} + \alpha_V L^V_{ALD} \tag{7.3}$$

where α_π and α_V are mixing coefficients to control the contribution of each loss.

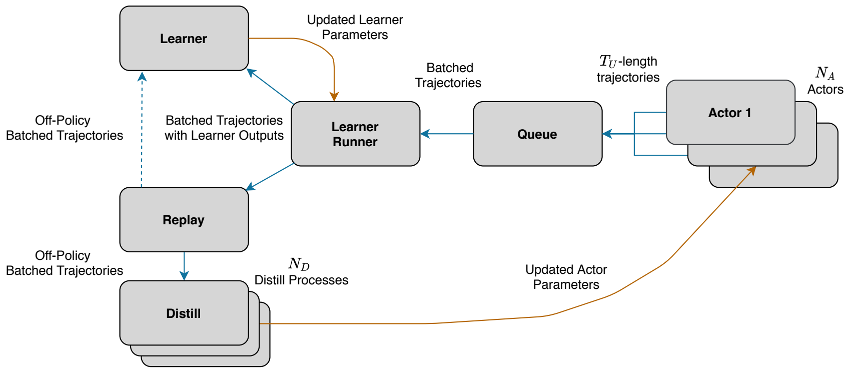

Learner
Updated Learner Parameters
Batched Trajectories
T_U-length trajectories
N_A Actors
Actor 1
Off-Policy Batched Trajectories
Batched Trajectories with Learner Outputs
Learner Runner
Queue
Replay
Off-Policy Batched Trajectories
N_D Distill Processes
Updated Actor Parameters
Distill

Figure 7.2: **Top:** An overview of distributed Actor-Learner Distillation, showing the processes as boxes and communication as arrows (data flow shown as blue arrows, parameter flow as orange).

7.3 Distributed Actor-Learner Distillation

In order to make more efficient use of computational resources, we develop a distributed system for Actor-Learner Distillation based on IMPALA [54]. An overview of this system is shown in Fig. 7.2, and we detail the function of each distinct process below, in the order of data flow:

Actor: There are N_A parallel actors, each executing on single-threaded processes with only CPU resources available. Each Actor process steps through environment steps sequentially until a trajectory of T_U time steps is gathered. Actions are sampled from a local copy of the actor policy π_A. Once a completed trajectory is acquired, it is communicated to a Queue process.

Queue: The Queue process receives trajectories asynchronously across actors and accumulates them into batches. The batched trajectories are then passed to the Learner Runner.

Learner Runner: The Learner Runner process runs $\mathcal{M}_L$ in inference mode on the incoming batches of data using a hardware accelerator. This is done to provide (1) learning targets to the Distill processes downstream and (2) to make sure learner model initial memory states are updated for the Learner. Specifically when $\mathcal{M}_L$ is a transformer model, we additionally batch over the time dimension to process data even more rapidly. The Learner Runner then passes the batched trajectories it received along with computed learner model outputs to two separate parallel processing streams, the Learner process and the Replay process.

Learner: The Learner process receives batched trajectories from the Learner Runner and computes, using a hardware accelerator, all relevant RL objectives (V-MPO in our case [188]), along with the loss in Eq. 7.1 to regularize the learner model towards the actor model. In the Learner process, only the learner model parameters are updated. Similarly to recent work on deep actor-critic algorithms [188, 127], we use a “target network” where we only communicate updated learner parameters to the Learner Runner every K_L optimization steps. Although not used here, the Learner process can additionally receive data from the Replay process.

Replay: The Replay process manages a replay buffer containing a large store of previously collected batched trajectories. Incoming batches of trajectories from the Learner Runner are archived in a large first-in-first-out queue. The Learner and Distill processes can then request batches of trajectories which are uniformly sampled from this queue.

Distill: Distill processes request data from the Replay buffer and use the retrieved trajectories to compute the distillation loss in Eq. 7.3, which is then used to update the actor model parameters. Similar to the Learner process, we utilize a target network scheme which updates model parameters on the Actor processes every K_A optimization steps.

7.3.1 Improving Distill / RL Step Ratio

An observation found early in the development of ALD was the significance of the "distillation steps per RL steps" (DpRL) ratio. The DpRL ratio measures how many agent steps are taken per second on the actor model (which takes "distillation steps") comparatively with the learner model’s agent steps per second (which takes "reinforcement learning steps"). This is exemplified in the left side of Fig. 7.3, where in the Meta-Fetch environment with 3 objects (see Sec. 7.5.2), we ran ALD but set a fixed number of Actor agent steps for every Learner agent step. The graph shows a greatly enhanced sample efficiency when the number of actor agent steps is increased in relation to

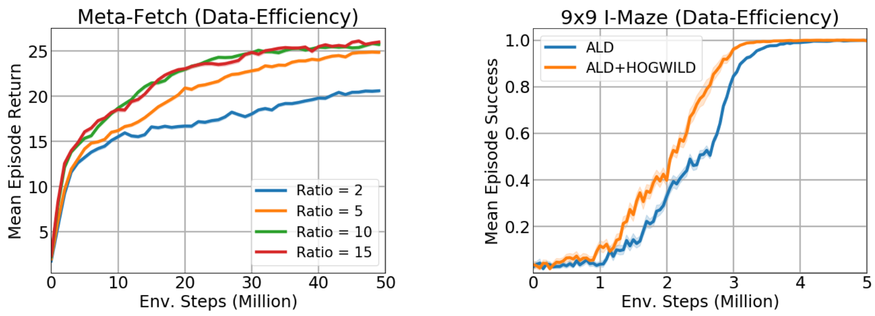


Figure 7.3: **Left:** On the Meta-Fetch environment with 3 objects, we evaluated the importance of the "Distillation steps per RL step" ratio. Shown on the left, as we increased this ratio we clearly observed an improved sample efficiency in the Actor-Learner Distillation procedure, until saturation at a ratio of 10. However, higher ratios came at the cost of substantial increase in wall-clock time, as the high number of distillation steps quickly becomes the bottleneck of the system's speed. **Right:** Parallelizing the distill processes using HOGWILD! improved the DpRL ratio without causing a slowdown in system speed. enabling an increase in data efficiency without a decrease in system speed, as shown in this result on 9x9 I-Maze.

learner agent steps, until saturating at around 10 actor agent steps per learner agent step. Out of all hyperparameters we tested, we found that a high DpRL ratio was consistently the most critical parameter in increasing the sample efficiency of ALD. However, increasing the DpRL ratio naturally increased the total run time of the system, as now the distillation process became the main system bottleneck, and this encouraged investigation into parallelized ways of improving actor agent steps.

To avoid a complete system slowdown while maintaining a high DpRL ratio, we leveraged the parallelized training procedure HOGWILD! [167] as a replacement to the single-Distill-process system originally tested. HOGWILD! enables lock-free stochastic gradient optimization to be performed by several parallel learning processes. Each process has shared access to the parameter vector of the learning model, and can update it without requiring a mutually-exclusive lock. This allows parameter updates to occur in parallel as soon as they are available, but potentially leads to race conditions where the parameter vector is read during it being updated by a different process. This means a fraction of the parameter vector can be stale when a learning process calculates the parameter gradients. However, while this parallel overwriting of the parameter vector introduces noise via this staleness, it usually does not significantly inhibit performance and is typically faster than alternatives which use explicit locking to prevent race conditions.

In the HOGWILD! ALD setting, there are now N_D parallel Distill processes, all of which asynchronously sample batched trajectories from the replay buffer to update actor model parameters. As there are typically more than 1 available accelerators on a single machine (see e.g. reference machines 7.4.3), the parallel Distill processes can be evenly distributed between available accelerators to make the best use of the resources at hand. In the right side of Figure 7.3, the HOGWILD! Actor-Learner Distillation variant consistently achieved better sample complexity with equivalent time complexity, across a wide sweep of hyper-parameter settings.

7.4 Methodology

7.4.1 Algorithm Details

For experiments, we use the V-MPO algorithm [188] as the RL algorithm underlying each of the procedures we test. For ALD, we use V-MPO with V-trace corrections [54] which worked slightly better in preliminary experiments. For baseline models V-MPO without V-trace worked better, as in [188]. For all experiments, we use a single-layer LSTM of varying dimension depending on the environment as the actor model. For transformers, we use the Gated Transformer-XL [155], which we initially observed had better sample efficiency and optimization stability than standard transformers. We vary the number of transformer layers dependent on the difficulty of the environment.

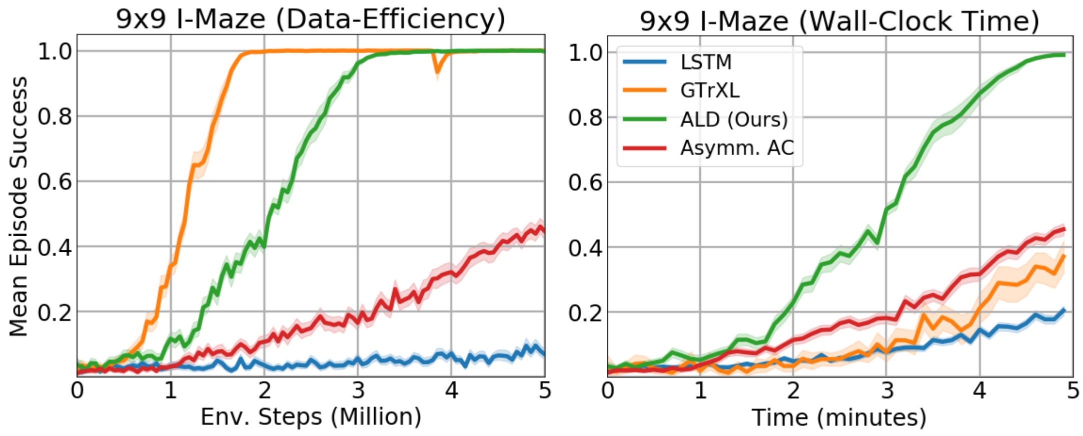


Figure 7.4: Results on the 9 × 9 I-Maze environment for all models. **Left:** x-axis as number of environment steps. **Right:** x-axis as wallclock time. All curves have 3 seeds. Obtained on reference machine A (Sec. 7.4.3).

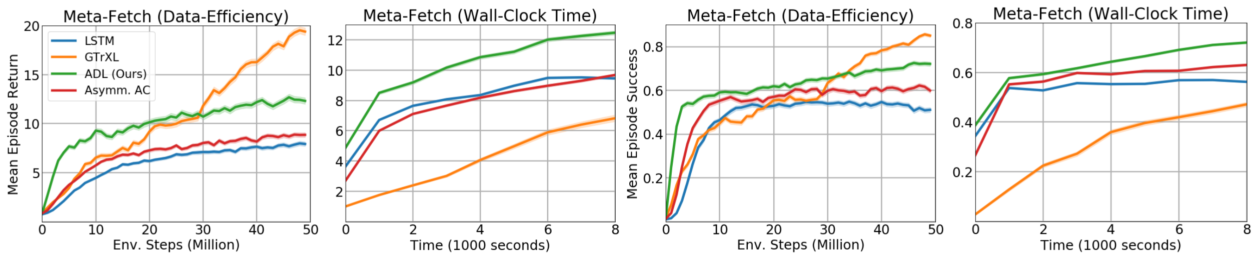


Figure 7.5: Results on the Meta-Fetch environment with 4 objects. **Left:** y-axis as environment returns (objects fetched correctly per episode). **Right:** y-axis as success rate (all 4 objects fetched correctly at least once in an episode). All curves have 3 seeds. Obtained on reference machine B (Section 7.4.3).

7.4.2 Baselines

For each environment, we individually run the actor and learner model architectures used in ALD as baselines. As an additional baseline besides the learner and actor models in isolation, we introduce a model which maintains a distinct policy and value network. Here the policy network has the same architecture as the actor model in ALD (i.e. a small LSTM) and the value network has the same architecture as the learner model (i.e. a transformer). As the value function does not need to be run on actor processes, only the policy, this baseline achieves similar run time as ALD without requiring any extra insights. This baseline is an instance of Asymmetric Actor-Critic (Asymm. AC) [159], but where instead of the value function having privileged access to state information it has privileged access to computational resources. Comparison of ALD to Asymm. AC will reveal whether the introduction of distillation losses has any positive effect over the simpler method of creating independent policy/value functions.

7.4.3 Compute Details

Reference Machine A: Reference Machine A has a 36-thread Intel(R) Core(TM) i9-7980XE CPU @ 2.60GHz, 64GB of RAM, and 2 GPUs: a GeForce GTX 1080 Ti and a TITAN V.

Reference Machine B: Reference Machine B has a 40-thread Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz, 256GB of RAM and 2 GPUs: a Tesla P40 and a Tesla V100-PCIE-16GB.

7.4.4 Common Experimental Details

For all models, we sweep over the V-MPO target network update frequency $K_L \in \{1, 10, 100\}$. In initial experiments, we also sweeped the "Initial α" setting over values $\{0.1, 0.5, 1.0, 5.0\}$. All experiment runs have 3 unique seeds. For each model, we choose the hyperparameter setting that achieved highest mean return over all seeds. Additionally we use PopArt [87] for the value output.

Hyperparameter	Value
Optimizer	Adam
Learning Rate	0.0001
N_A	30
N_D	8
Batch Size	64
T_U	20
Discount Factor (γ)	0.99
Grad. Norm. Clipping	Disabled
Initial η	1.0
Initial α	{0.1, 0.5, 1.0, 5.0}
ϵ_η	0.1
ϵ_α	0.004
K_L	{1, 10, 100}
PopArt Decay	0.0003

Table 7.1: Common hyperparameters across experiments.

7.5 Experiments

7.5.1 I-Maze

The I-Maze environment has the agent start at the top-left corner of a grid-world with the shape of an "I" (Fig. 7.6, Left). The agent must travel from the top-left corner to one of the bottom corners of the I, where the episode is ended and the agent receives reward. The particular corner the agent must travel to is revealed in the top-right corner of the I, which contains an "indicator" tile that is either 0 or 1 depending on whether the left or right bottom corner contains the reward. If the agent does not enter a terminating corner state in H time steps, the episode ends without a reward. The agent entering the correct bottom corner goal based on the indicator receives a reward of 1. Entering the incorrect corner results in the episode terminating with no reward. As memory is the main concern of these results instead of exploration, for larger maze results (i.e. a maze of width 15), we add a shaping reward of 0.01 whenever the agent visits a state along the main corridor of the "I". Once the agent has traversed a particular state, it cannot gain another shaping reward for returning there until the next episode. The agent has an orientation and observes every pixel directly in front of it until it reaches an occluding wall.

Observations, Actions and Metrics

Observations in I-Maze are the single row of pixels starting at the agent's current position and extending in the direction the agent is facing. The row of pixels extends as far as the maze dimension. If the agent has a wall in its field of view (represented as black pixels in Fig. 7.6), pixels further away from that point are occluded. For the 9×9 maze, a reward is only given when the agent enters the correct goal as decided by the indicator. For the 15×15 maze, it is the same but there is an additional shaping reward where whenever the agent enters one of the states along the central column of the "I" for the first time within its current episode, it receives a small reward of 0.01. This reward is meant to encourage exploration and prevent the very large number of environment steps otherwise necessary for the agent to end up at a goal when taking actions randomly. The agent has 4 actions, move forward, turn left, turn right and do nothing. Moving forward into a wall causes no change in state. 9×9 I-Maze episodes last for a horizon of 150 time steps while 15×15 I-Maze episodes last for 350 steps. Observations have 3 channel dimensions, one for whether the pixel is free space or a wall, one for the green goal indicator and one for the red goal indicator.

Architecture Details

LSTM: We use a single-layer LSTM with a hidden dimension of 32 for all experiments in this domain.
GTrXL: We use a 2-layer GTrXL for the experiments in 9×9 and a 4-layer GTrXL for the experiments in 9×9. We use an embedding size of 256, 8 attention heads, a head dimension of 32,

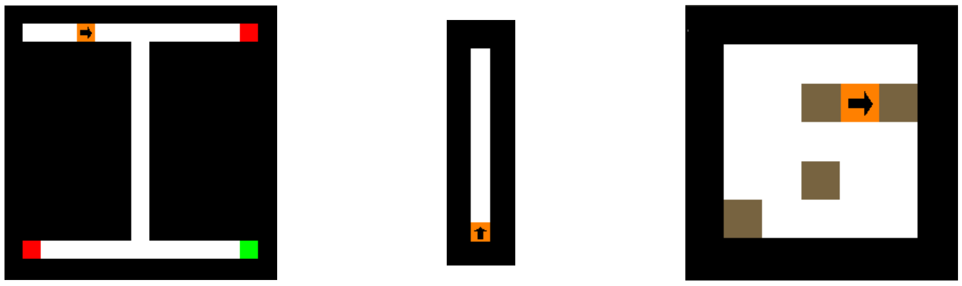


Figure 7.6: **Left:** Example of the 15×15-size I-Maze environment, with the indicator in this case being red. **Center:** An example of the agent observation, which is a 3-pixel-wide egocentric beam starting in the direction of the agent's orientation. The agent cannot see behind itself or through walls. **Right:** Example of the Meta-Fetch environment with 4 randomly located objects shown in brown. The agent is shown in orange with its orientation indicated by the direction of the arrow.

a gated initialization bias of 2 and a memory length of 64. Other details were followed from [155].
ALD: We use the corresponding LSTMs and GTrXL described above depending on the environment. During hyperparameter sweeps, we tested $K_A \in \{1, 10, 100\}$. We set $\alpha_\pi = 1$ and sweep $\alpha_V \in \{0, 0.1, 1\}$.
Asymm. AC: We use the corresponding LSTMs and GTrXL described above depending on the environment.

Discussion of Empirical Results

The environment provides a clear test on an agent's ability to remember events over long horizons, as the only way to reliably terminate an episode successfully is by remembering the indicator observed near the start of the maze. In Figure 7.4, we provide complete reward curves for a maze of dimension 9×9. We plot curves with two different x-axes: environment steps, measuring sample complexity, and wall-clock time. All curves were obtained on Reference Machine A (see Section 7.4.3) under identical operating conditions, meaning wall-clock time is a fair comparison. We can first observe the clear sample efficiency gains of the transformer over the LSTM as well as its poorer performance when considering wall-clock time. Furthermore, we can observe that Actor-Learner Distillation achieves significantly improved sample complexity over the stand-alone LSTM. This confirms the ability of ALD to recover the sample-efficiency gains of the learner model. In terms of wall-clock time, the Actor-Learner Distillation procedure is unmatched by any other procedure, demonstrating its clear practical advantage.

In contrast to Actor-Learner Distillation, the Asymm. AC baseline does not seem to perform as well despite equivalent model complexity and equivalent time complexity, as it contains both identical actor and learner models. To gain insight into this result, we plotted the per-seed curves of Actor-Learner Distillation and Asymm. AC on the 15×15 I-Maze task in Figure 7.7. We can clearly see an underlying pattern to the seed curves, all models have a local optimum at around 0.5 success where each model class spends a varying amount of time. A return of 0.5 suggests that the model has learnt to enter a goal, but not yet learnt to use the indicator to correctly determine which goal contains the positive reward. We measured how many environment steps each model class spent in this local minima out of the 10 million total, and found that the LSTM and Asymm. AC spend a majority of their time there. While it can be expected that the transformer can easily exit this optima due to its ability to directly look back in time, interestingly ALD seems to recover the same efficiency at escaping this minima as the stand-alone transformer. This suggests that ALD can successfully impart the learner model's inductive biases to the actor model during learning in a way that is substantially more effective than just using the transformer as a value function (Asymm. AC). Finally, in the left-hand-side of Fig. 7.7 we ran an ablation on 9×9 I-Maze to determine whether the value distillation loss provided benefit to learning. We performed a hyperparameter sweep for each setting of value distillation enabled or disabled. The results demonstrated that using value distillation provides a slight but significant improvement.

7.5.2 Meta-Fetch

The "Meta-Fetch" meta-reinforcement learning environment, shown in Figure 7.6, requires an agent to fetch a number of objects distributed randomly on an empty gridworld, with each object required

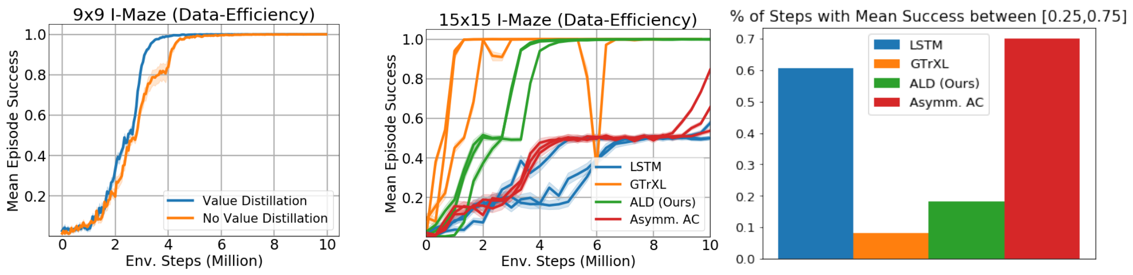


Figure 7.7: **Left:** Value distillation consistently provides a slight improvement to results. Results here were taken over a hyperparameter sweep with either value distillation enabled or disabled, and the best performing curves were chosen from each setting. **Center, Right:** Plotting per-seed curves (Center) of results in 15×15 I-Maze reveals that the Asymm. AC and LSTM baseline spend a significant amount of time achieving a mean success rate around 0.5 (quantified in the right-hand bar plot). A success of 0.5 suggests both models have learnt to enter a goal but are not yet able to use their memory to make the inference between goal entered and indicator identity. In contrast ALD and GTrXL rapidly learn to reach a perfect return of 1 and do not spend much time near 0.5.

to be fetched in a particular hidden sequence. Crucially, in Meta-Fetch (1) the agent can not sense the unique identity of each object, meaning each object looks the same to the agent, (2) the agent does not know beforehand the sequence of objects it must obtain, and (3) observations consist of local views of pixels in front of the agent's current orientation (see Fig. 7.6). When the agent collects an object, it can either receive a positive reward if this object is the next correct one in the sequence, or receive no reward and the sequence of objects is reset. To prevent cheating (i.e. the agent obtaining a reward and then immediately resetting the sequence, and repeating), the agent receives a positive reward only on the first instance of it collecting the next correct object in the sequence. Once all objects in the sequence have been fetched, the objects and rewards are reset and the environment proceeds as it did at the start of the episode (except the agent is now in the position of the last object fetched). Meta-Fetch presents a highly challenging problem for memory architectures because the agent must simultaneously learn the identity of objects by mapping their spatial relations using only local egocentric views, as well as learn a discrete search strategy to memorize the correct sequence of objects to hit. Meta-Fetch was inspired by the meta-learning environment "Number Pad" used in previous work [93, 155] and in Chapter 6.

Observations, Actions and Metrics

In Meta-Fetch, an agent acts in an empty 2D gridworld of size 7×7 given only local observations. At each step the agent can choose to either move forward or turn left or right. Observations are the 15 pixels in front of the agent pointing in it's orientation, along with the 2 pixels on either side of this ray, for a complete observation size of $C \times 15 \times 3$ pixels (see Fig. 7.6). Each observation has C channels which determine whether that pixel represents free space, a wall, an object, or a collected object. Every episode has the agent located in a new configuration of object locations. We set a maximum episode length of 300 steps for Meta-Fetch.

The goal of the agent in Meta-Fetch is to collect, by colliding with, each of the randomly located objects in a specific sequence. This sequence is not known beforehand and is randomized each episode. The only clue to discovering this sequence is that when the agent hits the next right object in the sequence, it gains a reward of 1 and the object is sensed as "collected". Otherwise, if it hit the wrong object in the sequence, then all previously collected objects are reset and the agent must restart the fetch sequence from the beginning once again. Once all objects are collected, they are all reset and the agent can restart the same sequence to once again gain reward for each collected reward. Objects which have been reset before the complete fetch sequence is achieved do not give further reward upon re-collection, to prevent the agent from falling into an local optimum of repeatedly collecting a correct object and then immediately collecting an incorrect object to reset the correct object.

Architecture Details

LSTM: We use a single-layer LSTM with a hidden dimension of 128.
GTrXL: We use a 4-layer GTrXL, an embedding size of 256, 8 attention heads, a head dimension

of 32, a gated initialization bias of 2 and a memory length of 64. Other details were followed from [155].
ALD: We use the corresponding LSTMs and GTrXL described above depending on the environment. During hyperparameter sweeps, we tested $K_A \in \{10, 100\}$. We set $\alpha_\pi = 1$ and sweep $\alpha_V \in \{0, 0.1, 1\}$.
Asymm. AC: We use the corresponding LSTMs and GTrXL described above depending on the environment.

Discussion of Empirical Results

Results are reported in Fig. 7.5 for a Meta-Fetch environment with 4 randomly located objects. We present plots of average return using both environment steps and wall-clock time as x-axes (left plots), along with an additional plot which measures "success" (right plots). Success in Meta-Fetch is defined as the agent completing at least 1 full object fetch sequence (agents can accomplish multiple full fetch sequences for additional reward). As in I-Maze, we reconfirm the observation that transformers achieve a superior sample efficiency over LSTMs, while at the cost of a much slower wall-clock performance. Unlike I-Maze where Asymm. AC had a more positive effect on sample efficiency, there is a less substantial difference between LSTM and Asymm. AC.

However, we can see a major difference between the baselines and Actor-Learner Distillation, which early on achieves close to the transformer's sample efficiency with a much smaller model. In this much more challenging environment, which requires both memory and compositional reasoning, the LSTM actor model in ALD suffers in performance later on when the GTrXL achieves a higher rate of increase in reward. Examining the success rate reveals an interesting trend where the ALD-trained LSTM achieves better generalization than the LSTM and Asymm. AC models. In particular, we can see that although the LSTM and Asymm. AC have average return that is increasing, their success rate is relatively stagnant (Asymm. AC) or even decreasing (LSTM). This suggests the increase in return these baselines are observing mainly stem from them learning how to search through a fraction of possible object layouts more efficiently. In contrast, the ALD-trained LSTM's success rate is correlated with its return, suggesting it is learning to succeed in a larger variety of object layouts.

7.6 Discussion

To conclude this chapter, towards the development of efficient learning methods in actor-latency-constrained settings, we developed the Actor-Learner Distillation (ALD) procedure, which leverages a continual form of model compression between separate actor and learner models. In the context of learning in partially-observable and meta-learning environments using a distributed actor-critic system, ALD successfully demonstrated the ability to largely capture the sample efficiency gains of a larger transformer learner model while still maintaining the reduced computational cost and lower experiment run time of the LSTM actor model. As supervised learning demonstrates ever increasing gains in performance from increasing model capacity, we believe the development of effective methods of model compression for RL will become a more prominent area of study in the future. Beyond increased scale, we hope that the reduced computational costs of training transformers using ALD can unlock the use of transformers more widely.

With the close of this chapter and, further, this thesis section, we have described a methodology which leverages powerful, scalable and general sequence models in order to accomplish few-shot learning in reinforcement learning domains. Additionally, we have detailed novel parallelized execution settings for meta-learning agents as well as a distributed reinforcement learning system that uses distillation to significantly reduce the computational resources required to train these models, with the goal of democratizing their more widespread use within the research community as well as allowing them to be applied to latency-constrained environments. In the next chapter, we will conclude this thesis by first re-iterating the goals we set out in the introduction, and then summarizing the main achievements towards these goals accomplished within the chapters. Finally, connections to on-going and future work will be highlighted and discussed.

Chapter 8

Conclusion

While reinforcement learning has demonstrated increasingly more impactful capabilities in many domains, its applicability to many real-world settings remains impeded by the high sample complexity of the learning algorithms, which can require millions of interactions with a system in order to gain competency. While there are numerous reasons underpinning the data inefficiency of these algorithms, such as inefficient exploration algorithms, this poor sample complexity is in major part owed to the generality of environment types these algorithms can be applied to, essentially covering any setting that can be formulated as a Markov Decision Process. Specializing the domain of environments an algorithm is applied to can enable more efficient learning by exposing novel inductive biases that leverage statistical structure within the target task distribution. However, hand-designing such inductive biases for each class of tasks of interest could require substantial domain-knowledge and experimentation, making the cost of such specialization prohibitive.

The promise of meta-learning, or learning how to learn, is to automatically specialize a learning algorithm to a set of environment domains, by the learning process itself using a nested set of learning algorihms. More specifically, meta-learning algorithms are structured into a slow outer-loop and a fast, efficient inner-loop of learning: the inner-loop of learning is meant to be specialized to a set of related target tasks, which ideally share some underlying structure that can be exploited, while the outer-loop of learning adapts some set of inner-loop optimizable parameters to the target task distribution. The outer-loop learning algorithm is typically a more general, i.e. unspecialized, one, such as a reinforcement learning or self-supervised learning algorithm. The outer-loop adjusts the parameters of the inner-loop over a set of training tasks sampled from the target task distribution, enabling the inner-loop learning algorithm to adapt its own inductive biases to quickly learn any new task sampled from this task distribution. Meta-learning therefore amortizes the data inefficiency of training the inner-loop over the entire target task distribution, promising a tuned inner-loop learning algorithm that can solve any new tasks from the same distribution with much fewer data.

In this thesis, we focused on the development of meta-learning techniques which used parameterized deep sequence models as the inner-loop learning algorithm. In this setting, an agent has a memory architecture whose temporal receptive field spans across multiple episodes of interaction of the agent with a particular task or distribution of tasks. By being fed in as input the actions and rewards in addition to the states, the sequence model is able to act as its own learning algorithm, adjusting its internal representations to effectively decide to e.g. target new areas of the state space to explore or to maximize its return on the next episode of interaction. By relying on a sequence model to perform rapid adaptation to new tasks drawn from the target distribution, this method functionally reduces the design of meta-learning algorithms into one of architecture design.

We presented several methods for meta-learning through memory. Section 1 exploited the structural regularities available in embodied environments, an important setting which encompasses robotics, in the design of novel memory models. Within the three chapters of this first section, we built out the individual components of a monolithic embodied agent architecture: first, in Chapter 2, a map-structured memory for the agent to rapidly retain semantic information about its surroundings; second, in Chapter 3, architectures designed for enabling planning and state-inference capabilities in the agent; and, finally, in Chapter 4, an architecture capable of simultaneously mapping its environment and localizing itself with its partially-built map.

Section 2 then moved onto more general settings which did not make strong assumptions about the particular environment structure. Chapter 5 introduced a novel parallelized execution strategy for meta-learning through memory agents, which stands in contrast to the sequential episode processing typically used. Parallelized execution strategies are particularly important when environment interaction is slow but can be parallelized, as it can increase the total experience throughput of the meta-learning agent. In Chapter 6, by leveraging recent advancements in attention-based sequence models we developed the gated transformer, a modified transformer architecture with improved optimization stability that demonstrated state-of-the-art performance on DMLab-30. Finally, in Chapter 7, we presented Actor-Learner Distillation, a distributed reinforcement learning system that used an online form of distillation to significantly reduce the substantial computational resources required for training transformer-style models, with the goal of democratizing these models for the wider community as well as unlocking their application to latency-constrained settings, such as deployment onboard a robotic platform.

8.1 Future Directions

There are many possible future avenues for research within the space of memory-based meta-reinforcement learning. Here we note a few of them of particular interest, especially those which are logical extensions of the work presented in this thesis.

8.1.1 Scaling Meta-Learning through Memory: Model Capacity

There has recently been a consistent demonstration across many different domains that, as transformers are scaled in model capacity and as datasets grow increasingly larger, a significant and profound ability to generalize to new tasks within the domain emerges [101, 83, 19]. In this setting, a massive transformer is pretrained on vast amounts of data using simple self-supervised losses and is then applied to new domains that have some similarity to data seen during pretraining. Given only a few examples, which are used to either finetune the weights of the model or condition the transformer using its attention context, the pretrained model can then often achieve state-of-the-art performance on the downstream task [19]. Notably this ability to generalize continues to improve as both the pretraining dataset size and model scale are increased, even when the model is scaled to use entire datacenters worth of hardware accelerators [101, 83].

Therefore by amortizing the cost of pretraining a very large model over vast amounts of cheaply available data, these models then gain a significant data-efficiency at downstream tasks where larger amounts of data are likely unavailable. Given that the generalization ability of this process has been validated in a variety of domains [19, 27, 83], it remains highly plausible that a similar sort of generalization can emerge in control. One could imagine a single large transformer, e.g. on the parameter scale of the GPT-2 language model [164], trained on an extremely large set of tasks, and potentially demonstrating the ability to control disparate embodiments (e.g. different manipulation or locomotion robots) or separate environment domains (e.g. different video games). It is also possible that such a model could then be further finetuned to control novel embodiments or tasks than the ones seen during training. If such a model could retain the data efficiency gains seen by pretraining in other domains, it could result in a completely new approach to solving control problems, and could open up the application of reinforcement learning agents to completely new fields where data is extremely sparse or expensive to collect.

8.1.2 Scaling Meta-Learning through Memory: Context Length

Beyond the ability of large transformers to generalize to new tasks given small amounts of training data, it has also been observed that, when given enough model capacity, large language models can solve tasks simply by being conditioned on the type of behaviour that is desired [19]. For example, given a dataset in the form of N $\{(x_i, y_i)\}_{i=1}^N$ pairs and a prediction input $\hat{x}$, one could feed in the data directly as a string such as "$(x_1, y_1), \ldots, (x_N, y_N), (\hat{x},$" and then sample from the model to get the continuation of the string (and therefore the desired $\hat{y}$ prediction). Through this process of "prompting" the model, a surprising amount of knowledge can be successfully extracted from these large models without any changes to their weights.

The connections between prompting and the multi-episode meta-learning that was the focus of this thesis are clear, as the prompts can be seen to constitute a hand-constructed multi-episode supervised learning task. Therefore the ability of the large-scale language models to be able to perform prompt conditioning without being explicitly trained to do so, and to further generalize to types of prompts not seen explicitly in the training data, provides a significant motivation to explore a similar mechanism for control domains. In this setting, a transformer could conceivably be conditioned on a large store of interaction experience with a particular environment instance, essentially a small replay buffer of experience, and then produce an optimal behaviour conditioned on that data.

There are many possibilities on how to formulate this conditioning mechanism: is the data being conditioned on obtained from random policies? Or from optimal policies, i.e. demonstrations? Or is it from the transformer itself interacting with the environment, i.e. RLˆ2 [49]? Furthermore, there are many avenues to explore to find learning objectives from which successful prompting behaviour in the control setting could emerge – would simple objectives like reinforcement learning with long enough context windows enable this kind of generalization? Alternatively, it could be that prompting must be folded more directly into the training methodology, for example by sampling plausible prompts during pretraining. Given the large differences in compositionality between language and control, it is plausible that more speciailized losses or training procedures could be required. Although there are many unanswered questions, this type of conditional prompt-based generalization would represent the pinnacle achievement of the meta-learning through memory paradigm and would have revolutionary impact on practical domains, as such a model could be feasibly applied to few-shot learning scenarios.

8.1.3 Improved Efficiency for Large-Scale Transformers

As discussed in Chapter 7 , related to the research direction of scaling meta-learning models, an important aspect of applying transformers to reinforcement learning is inference speed. As agent model scale increases to the multi-billion parameter models seen in supervised domains like language modeling [164, 19], inference will be increasingly the main computational bottleneck as environment interaction will require a complete forward pass of the network for every action taken. Furthermore, unlike in supervised settings, in reinforcement learning model compression cannot be performed effectively offline and requires the learning model to interact with the environment in some capacity.

Therefore when pre-training the types of large-scale transformers as discussed in the previous sections, the issue of policy inference will likely come to the forefront in a way it has not been previously in reinforcement learning, which has typically used smaller models compared to supervised domains. The question of how to do efficient data collection while maintaining a certain degree of on-policyness will become a core issue, and while Actor-Learner Distillation as described in Chapter 7 is one possible solution, it has not been comprehensively validated at larger scale, which can present new and unforeseen issues that were not readily apparent at smaller scale. Even if pretraining methods for control could somehow be done completely offline, e.g. using some form of offline reinforcement learning, the issues relating to data collection with an expensive model are thus only pushed downstream: after pretraining, some type of data collection will likely need to occur on the target task in order for finetuning to be possible.

Research into efficient policy inference is also not constrained to the more narrow setting of large pretrained transformer models and is likely to have wider impact on the general field of reinforcement learning. As agent architectures scale to larger sizes due to increasing environment complexity, more attention will need to be paid on how data collection and policy inference can occur without incurring excessive experiment runtimes or computational cost. Additionally in actor-latency constrained domains, where acting is under hard constraints on action sampling latency, achievable model capacity is directly linked with policy inference speed as acting is usually done on limited or unaccelerated hardware, e.g. onboard a mobile robotic platform.

8.1.4 General Sequence Models for Embodied Environments

In the first half of this thesis, we presented a path towards a monolithic architecture for meta-learning in embodied environments. This agent architecture consisted of several subcomponents which mimic the traditional elements typically seen in Simultaneous Localization and Mapping (SLAM) systems [140, 139], i.e. a map, a (re-)localizer module, a path planner, and a graph-based optimizer. Each of these subcomponent architectures significantly benefited in their design from an existing understanding of the structure present in embodied environments, and while they are arguably less intensive in their requirement of domain expertise compared to traditional methods, they are still derived from a large amount of prior knowledge. In contrast, a fully end-to-end trainable system would minimize the amount of domain knowledge required, and with enough available training data the more general models could feasibly achieve better performance.

Motivated by this, Section 2 of this thesis switched focus to research the application of powerful and more general sequence models to the domain of meta-reinforcement learning through memory. However, there is missing a more concentrated examination of the general sequence models described in Section 2, like the gated transformer, to embodied environment tasks and benchmarks. A future direction of work is then to determine whether the weaker domain-specific inductive biases of the transformer-based models can achieve better performance than the more hand-crafted architectures that are currently used in these environments. Given the results of Chapter 6 in DMLab-30's navigation levels as well as recent contemporary work [55], there is an emerging body of evidence that is showing support to the hypothesis that the newer general sequence models like transformers could be more effective in embodied domains than the alternatives.

Bibliography

[1] Abbas Abdolmaleki, Jost Tobias Springenberg, Jonas Degrave, Steven Bohez, Yuval Tassa, Dan Belov, Nicolas Heess, and Martin Riedmiller. Relative Entropy Regularized Policy Iteration. *arXiv preprint*, 2018.

[2] Abbas Abdolmaleki, Jost Tobias Springenberg, Yuval Tassa, Remi Munos, Nicolas Heess, and Martin Riedmiller. Maximum a Posteriori Policy Optimisation. *Int. Conf. Learn. Represent.*, 2018.

[3] Abbas Abdolmaleki, Jost Tobias Springenberg, Yuval Tassa, Remi Munos, Nicolas Heess, and Martin Riedmiller. Maximum a posteriori policy optimisation. In *International Conference on Learning Representations*, 2018.

[4] Samira Abnar, Mostafa Dehghani, and Willem Zuidema. Transferring inductive biases through knowledge distillation. *arXiv preprint arXiv:2006.00555*, 2020.

[5] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in neural information processing systems*, pages 3981–3989, 2016.

[6] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In *Advances in neural information processing systems*, pages 2654–2662, 2014.

[7] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[8] Alexei Baevski and Michael Auli. Adaptive input representations for neural language modeling. *Int. Conf. Learn. Represent.*, 2019.

[9] D. Bahdanau, K. H. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *Proceedings of the 3rd International Conference on Learning Representations, ICLR*, 2015.

[10] Charles Beattie, Joel Z Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, et al. Deepmind lab. *arXiv preprint arXiv:1612.03801*, 2016.

[11] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[12] Glen Berseth, Cheng Xie, Paul Cernek, and Michiel Van de Panne. Progressive reinforcement learning with distillation for multi-skilled motion control. In *International Conference on Learning Representations*, 2018.

[13] Shehroze Bhatti, Alban Desmaison, Ondrej Miksik, Nantas Nardelli, N Siddharth, and Philip HS Torr. Playing doom with slam-augmented deep reinforcement learning. *arXiv preprint arXiv:1612.00380*, 2016.

[14] Charles Blundell, Benigno Uria, Alexander Pritzel, Yazhe Li, Avraham Ruderman, Joel Z Leibo, Jack Rae, Daan Wierstra, and Demis Hassabis. Model-free episodic control. *arXiv preprint arXiv:1606.04460*, 2016.

[15] Charles Blundell, Benigno Uria, Alexander Pritzel, Yazhe Li, Avraham Ruderman, Joel Z Leibo, Jack Rae, Daan Wierstra, and Demis Hassabis. Model-free episodic control. *arXiv preprint arXiv:1606.04460*, 2016.

[16] Mathew Botvinick, Sam Ritter, Jane X Wang, Zeb Kurth-Nelson, Charles Blundell, and Demis Hassabis. Reinforcement learning, fast and slow. *Trends in cognitive sciences*, 2019.

[17] Sean L Bowman, Nikolay Atanasov, Kostas Daniilidis, and George J Pappas. Probabilistic data association for semantic slam. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 1722–1729. IEEE, 2017.

[18] Eric Brachmann, Alexander Krull, Sebastian Nowozin, Jamie Shotton, Frank Michel, Stefan Gumhold, and Carsten Rother. Dsac-differentiable ransac for camera localization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6684–6692, 2017.

[19] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[20] Peter Buchlovsky, David Budden, Dominik Grewe, Chris Jones, John Aslanides, Frederic Besse, Andy Brock, Aidan Clark, Sergio Gomez Colmenarejo, Aedan Pope, Fabio Viola, and Dan Belov. TF-Replicator: Distributed Machine Learning for Researchers. *arXiv preprint*, 2019.

[21] Wolfram Burgard, Andrcas Derr, Dieter Fox, and Armin B Cremers. Integrating global position estimation and position tracking for mobile robots: the dynamic markov localization approach. In *Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on*, volume 2, pages 730–735. IEEE, 1998.

[22] Wolfram Burgard, Dieter Fox, Daniel Hennig, and Timo Schmidt. Estimating the absolute position of a mobile robot using position probability grids. In *Proceedings of the national conference on artificial intelligence*, pages 896–901, 1996.

[23] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, José Neira, Ian Reid, and John J Leonard. Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Transactions on robotics*, 32(6):1309–1332, 2016.

[24] Devendra Singh Chaplot, Guillaume Lample, Kanthashree Mysore Sathyendra, and Ruslan Salakhutdinov. Transfer deep reinforcement learning in 3d environments: An empirical study. In *NIPS Deep Reinforcemente Leaning Workshop*, 2016.

[25] Devendra Singh Chaplot, Emilio Parisotto, and Ruslan Salakhutdinov. Active neural localization. In *International Conference on Learning Representations*, 2018.

[26] Devendra Singh Chaplot, Emilio Parisotto, and Ruslan Salakhutdinov. Active neural localization. In *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018)*, 2018.

[27] Mark Chen, Alec Radford, Rewon Child, Jeffrey Wu, Heewoo Jun, David Luan, and Ilya Sutskever. Generative pretraining from pixels. In *International Conference on Machine Learning*, pages 1691–1703. PMLR, 2020.

[28] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[29] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[30] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[31] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[32] Luo Chunjie, Yang Qiang, et al. Cosine normalization: Using cosine similarity instead of dot product in neural networks. *arXiv preprint arXiv:1702.05870*, 2017.

[33] Ronald Clark, Sen Wang, Andrew Markham, Niki Trigoni, and Hongkai Wen. Vidloc: 6-dof video-clip relocalization. *arXiv preprint arXiv:1702.06521*, 2017.

[34] Ronald Clark, Sen Wang, Hongkai Wen, Andrew Markham, and Niki Trigoni. Vinet: Visual-inertial odometry as a sequence-to-sequence learning problem. In *AAAI*, pages 3995–4001, 2017.

[35] Karl Cobbe, Jacob Hilton, Oleg Klimov, and John Schulman. Phasic policy gradient. *arXiv preprint arXiv:2009.04416*, 2020.

[36] Gabriele Costante, Michele Mancini, Paolo Valigi, and Thomas A Ciarfuglia. Exploring representation learning with cnns for frame-to-frame ego-motion estimation. *IEEE robotics and automation letters*, 1(1):18–25, 2015.

[37] Ingemar J Cox and John J Leonard. Modeling a dynamic environment using a bayesian multiple hypothesis approach. *Artificial Intelligence*, 66(2):311–344, 1994.

[38] Wojciech Czarnecki, Siddhant Jayakumar, Max Jaderberg, Leonard Hasenclever, Yee Whye Teh, Nicolas Heess, Simon Osindero, and Razvan Pascanu. Mix & match agent curricula for reinforcement learning. In *International Conference on Machine Learning*, pages 1087–1095, 2018.

[39] Zihang Dai, Guokun Lai, Yiming Yang, and Quoc V Le. Funnel-transformer: Filtering out sequential redundancy for efficient language processing. *arXiv preprint arXiv:2006.03236*, 2020.

[40] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2978–2988, Florence, Italy, July 2019. Association for Computational Linguistics.

[41] Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 933–941. JMLR. org, 2017.

[42] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. *Int. Conf. Learn. Represent.*, 2018.

[43] Frank Dellaert. Factor graphs and gtsam: A hands-on introduction. Technical report, Georgia Institute of Technology, 2012.

[44] Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. Deep image homography estimation. *arXiv preprint arXiv:1606.03798*, 2016.

[45] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[46] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[47] Maria Dimakopoulou, Ian Osband, and Benjamin Van Roy. Scalable coordinated exploration in concurrent reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 4223–4232, 2018.

[48] Maria Dimakopoulou and Benjamin Van Roy. Coordinated exploration in concurrent reinforcement learning. In *International Conference on Machine Learning*, pages 1270–1278, 2018.

[49] Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

[50] Yadin Dudai and Mary Carruthers. The janus face of mnemosyne. *Nature*, 434(7033):567–567, 2005.

[51] Sergey Edunov, Myle Ott, Michael Auli, and David Grangier. Understanding back-translation at scale. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 489–500, 2018.

[52] Jakob Engel, Thomas Schöps, and Daniel Cremers. Lsd-slam: Large-scale direct monocular slam. In *European Conference on Computer Vision*, pages 834–849. Springer, 2014.

[53] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International Conference on Machine Learning*, pages 1406–1415, 2018.

[54] Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *ICML*, pages 1406–1415, 2018.

[55] Kuan Fang, Alexander Toshev, Li Fei-Fei, and Silvio Savarese. Scene memory transformer for embodied agents in long-horizon tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 538–547, 2019.

[56] C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, pages 1126–1135, 2017.

[57] Philipp Fischer, Alexey Dosovitskiy, Eddy Ilg, Philip Häusser, Caner Hazırbaş, Vladimir Golkov, Patrick Van der Smagt, Daniel Cremers, and Thomas Brox. Flownet: Learning optical flow with convolutional networks. *arXiv preprint arXiv:1504.06852*, 2015.

[58] Martin A Fischler and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[59] Sebastian Flennerhag, Andrei A. Rusu, Razvan Pascanu, Francesco Visin, Hujun Yin, and Raia Hadsell. Meta-learning with warped gradient descent. In *International Conference on Learning Representations*, 2020.

[60] Jakob Foerster, Ioannis Alexandros Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 2137–2145, 2016.

[61] Christian Forster, Matia Pizzoli, and Davide Scaramuzza. Svo: Fast semi-direct monocular visual odometry. In *2014 IEEE international conference on robotics and automation (ICRA)*, pages 15–22. IEEE, 2014.

[62] Christian Forster, Zichao Zhang, Michael Gassner, Manuel Werlberger, and Davide Scaramuzza. Svo: Semidirect visual odometry for monocular and multicamera systems. *IEEE Transactions on Robotics*, 33(2):249–265, 2016.

[63] Dieter Fox. *Markov localization-a probabilistic framework for mobile robot localization and navigation.* PhD thesis, Universität Bonn, 1998.

[64] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. Active markov localization for mobile robots. *Robotics and Autonomous Systems*, 25(3-4):195–207, 1998.

[65] V Fox, Jeffrey Hightower, Lin Liao, Dirk Schulz, and Gaetano Borriello. Bayesian filtering for location estimation. *IEEE pervasive computing*, 2(3):24–33, 2003.

[66] Tommaso Furlanello, Zachary Lipton, Michael Tschannen, Laurent Itti, and Anima Anandkumar. Born again neural networks. In *International Conference on Machine Learning*, pages 1607–1616, 2018.

[67] Dorian Gálvez-López and Juan D Tardos. Bags of binary words for fast place recognition in image sequences. *IEEE Transactions on Robotics*, 28(5):1188–1197, 2012.

[68] Dibya Ghosh, Jad Rahme, Aviral Kumar, Amy Zhang, Ryan P Adams, and Sergey Levine. Why generalization in rl is difficult: Epistemic pomdps and implicit partial observability. *arXiv preprint arXiv:2107.06277*, 2021.

[69] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[70] Google. Cloud TPU, 2018.

[71] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, A. P. Badia, K. M. Hermann, Y. Zwols, G. Ostrovski, A. Cain, H. King, C. Summerfield, P. Blunsom, K. Kavukcuoglu, and D. Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538:471–476, 2016.

[72] Alex Graves. Generating sequences with recurrent neural networks, 2013.

[73] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471, 2016.

[74] S. Gupta, J. Davidson, S. Levine, R. Sukthankar, and J. Malik. Cognitive mapping and planning for visual navigation. *CoRR*, abs/1702.03920, 2017.

[75] Saurabh Gupta, James Davidson, Sergey Levine, Rahul Sukthankar, and Jitendra Malik. Cognitive mapping and planning for visual navigation. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2017)*, pages 7272–7281, 2017.

[76] Saurabh Gupta, James Davidson, Sergey Levine, Rahul Sukthankar, and Jitendra Malik. Cognitive mapping and planning for visual navigation. *arXiv preprint arXiv:1702.03920*, 2017.

[77] Saurabh Gupta, David F. Fouhey, Sergey Levine, and Jitendra Malik. Unifying map and landmark based representations for visual navigation. *CoRR*, abs/1712.08125, 2017.

[78] Tuomas Haarnoja, Anurag Ajay, Sergey Levine, and Pieter Abbeel. Backprop kf: Learning discriminative deterministic state estimators. In *Advances in Neural Information Processing Systems*, pages 4376–4384, 2016.

[79] Demis Hassabis. *Neural processes underpinning episodic memory*. PhD thesis, UCL (University College London), 2009.

[80] M. Hausknecht and P. Stone. Deep recurrent q-learning for partially observable mdps. In *Proceedings of Conference on Artificial Intelligence, AAAI*, 2015.

[81] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[82] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016.

[83] Tom Henighan, Jared Kaplan, Mor Katz, Mark Chen, Christopher Hesse, Jacob Jackson, Heewoo Jun, Tom B Brown, Prafulla Dhariwal, Scott Gray, et al. Scaling laws for autoregressive generative modeling. *arXiv preprint arXiv:2010.14701*, 2020.

[84] Joao F Henriques and Andrea Vedaldi. Mapnet: An allocentric spatial memory for mapping environments. In *proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8476–8484, 2018.

[85] Joel A Hesch, Dimitrios G Kottas, Sean L Bowman, and Stergios I Roumeliotis. Camera-imu-based localization: Observability analysis and consistency improvement. *The International Journal of Robotics Research*, 33(1):182–201, 2014.

[86] Matteo Hessel, Hubert Soyer, Lasse Espeholt, Wojciech Czarnecki, Simon Schmitt, and Hado van Hasselt. Multi-task Deep Reinforcement Learning with PopArt. *arXiv preprint*, 2018.

[87] Matteo Hessel, Hubert Soyer, Lasse Espeholt, Wojciech Czarnecki, Simon Schmitt, and Hado van Hasselt. Multi-task deep reinforcement learning with popart. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3796–3803, 2019.

[88] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[89] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[90] Yedid Hoshen. Vain: Attentional multi-agent predictive modeling. In *Advances in Neural Information Processing Systems*, pages 2701–2711, 2017.

[91] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. Learning deep structured semantic models for web search using clickthrough data. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, pages 2333–2338. ACM, 2013.

[92] Jan Humplik, Alexandre Galashov, Leonard Hasenclever, Pedro A. Ortega, Yee Whye Teh, and Nicolas Heess. Meta reinforcement learning as task inference. *arXiv preprint*, 2019.

[93] Jan Humplik, Alexandre Galashov, Leonard Hasenclever, Pedro A Ortega, Yee Whye Teh, and Nicolas Heess. Meta reinforcement learning as task inference. *arXiv preprint arXiv:1905.06424*, 2019.

[94] Max Jaderberg, Wojciech M Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio Garcia Castaneda, Charles Beattie, Neil C Rabinowitz, Ari S Morcos, Avraham Ruderman, et al. Human-level performance in 3d multiplayer games with population-based reinforcement learning. *Science*, 364(6443):859–865, 2019.

[95] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017.

[96] Patric Jensfelt and Steen Kristensen. Active global localization for a mobile robot using multiple hypothesis tracking. *IEEE Transactions on Robotics and Automation*, 17(5):748–760, 2001.

[97] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998.

[98] Michael Kaess, Hordur Johannsson, Richard Roberts, Viorela Ila, John J Leonard, and Frank Dellaert. isam2: Incremental smoothing and mapping using the bayes tree. *The International Journal of Robotics Research*, 31(2):216–235, 2012.

[99] Nal Kalchbrenner, Ivo Danihelka, and Alex Graves. Grid long short-term memory. 07 2015.

[100] Rudolph Emil Kalman et al. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960.

[101] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[102] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. *Int. Conf. Learn. Represent.*, 2019.

[103] Peter Karkus, David Hsu, and Wee Sun Lee. QMDP-Net: Deep learning for planning under partial observability. In *Advances in Neural Information Processing Systems (NIPS 2017)*, pages 4697–4707, 2017.

[104] Peter Karkus, David Hsu, and Wee Sun Lee. Qmdp-net: Deep learning for planning under partial observability. *CoRR*, abs/1703.06692, 2017.

[105] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. ViZDoom: A Doom-based AI research platform for visual reinforcement learning. In *IEEE Conference on Computational Intelligence and Games*, pages 341–348, Santorini, Greece, Sep 2016. IEEE.

[106] Alex Kendall, Matthew Grimes, and Roberto Cipolla. Posenet: A convolutional network for real-time 6-dof camera relocalization. In *Proceedings of the IEEE international conference on computer vision*, pages 2938–2946, 2015.

[107] Arbaaz Khan, Clark Zhang, Nikolay Atanasov, Konstantinos Karydis, Vijay Kumar, and Daniel D Lee. Memory augmented control networks. *arXiv preprint arXiv:1709.05706*, 2017.

[108] Dimitrios G Kottas, Joel A Hesch, Sean L Bowman, and Stergios I Roumeliotis. On the consistency of vision-aided inertial navigation. In *Experimental Robotics*, pages 303–317. Springer, 2013.

[109] Ben Krause, Liang Lu, Iain Murray, and Steve Renals. Multiplicative lstm for sequence modelling, 2016.

[110] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[111] Rainer Kümmerle, Giorgio Grisetti, Hauke Strasdat, Kurt Konolige, and Wolfram Burgard. g 2 o: A general framework for graph optimization. In *2011 IEEE International Conference on Robotics and Automation*, pages 3607–3613. IEEE, 2011.

[112] Adhiguna Kuncoro, Chris Dyer, Laura Rimell, Stephen Clark, and Phil Blunsom. Scalable syntax-aware language models using knowledge distillation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3472–3484, 2019.

[113] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. *arXiv preprint arXiv:1609.05521*, 2016.

[114] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. In *AAAI*, pages 2140–2146, 2017.

[115] Guillaume Lample, Alexandre Sablayrolles, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Large memory layers with product keys. In *Advances in Neural Information Processing Systems*, pages 8548–8559, 2019.

[116] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[117] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[118] Yann LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, pages 9–50, London, UK, UK, 1998. Springer-Verlag.

[119] Lisa Lee, Emilio Parisotto, Devendra Singh Chaplot, Eric Xing, and Ruslan Salakhutdinov. Gated path planning networks. In *International Conference on Machine Learning*, pages 2953–2961, 2018.

[120] Máté Lengyel and Peter Dayan. Hippocampal contributions to control: the third way. In *Advances in neural information processing systems*, pages 889–896, 2008.

[121] Stefan Leutenegger, Paul Furgale, Vincent Rabaud, Margarita Chli, Kurt Konolige, and Roland Siegwart. Keyframe-based visual-inertial slam using nonlinear optimization. *Proceedings of Robotis Science and Systems (RSS) 2013*, 2013.

[122] Michael L Littman. The witness algorithm: Solving partially observable markov decision processes. 1994.

[123] Michael L Littman, Anthony R Cassandra, and Leslie Pack Kaelbling. Learning policies for partially observable environments: Scaling up. In *Machine Learning Proceedings 1995*, pages 362–370. Elsevier, 1995.

[124] Fayao Liu, Chunhua Shen, and Guosheng Lin. Deep convolutional neural fields for depth estimation from a single image. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5162–5170, 2015.

[125] Peter J Liu, Mohammad Saleh, Etienne Pot, Ben Goodrich, Ryan Sepassi, Lukasz Kaiser, and Noam Shazeer. Generating wikipedia by summarizing long sequences. *Int. Conf. Learn. Represent.*, 2018.

[126] Yang Liu and Mirella Lapata. Text summarization with pretrained encoders. *Conference on Empirical Methods in Natural Language Processing*, 2019.

[127] Michael Luo, Jiahao Yao, Richard Liaw, Eric Liang, and Ion Stoica. Impact: Importance weighted asynchronous architectures with clipped target networks. In *International Conference on Learning Representations*, 2020.

[128] Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In *International Conference on Machine Learning*, pages 2113–2122, 2015.

[129] Maze Generation Algorithms. Maze generation algorithms — Wikipedia, the free encyclopedia, 2018. [Online; accessed 9-Feb-2018].

[130] John McCormac, Ankur Handa, Andrew Davison, and Stefan Leutenegger. Semanticfusion: Dense 3d semantic mapping with convolutional neural networks. In *2017 IEEE International Conference on Robotics and automation (ICRA)*, pages 4628–4635. IEEE, 2017.

[131] Iaroslav Melekhov, Juha Ylioinas, Juho Kannala, and Esa Rahtu. Relative camera pose estimation using convolutional neural networks. In *International Conference on Advanced Concepts for Intelligent Vision Systems*, pages 675–687. Springer, 2017.

[132] Thomas Miconi, Kenneth Stanley, and Jeff Clune. Differentiable plasticity: training plastic neural networks with backpropagation. In *International Conference on Machine Learning*, pages 3556–3565, 2018.

[133] Piotr Mirowski, Razvan Pascanu, Fabio Viola, Hubert Soyer, Andy Ballard, Andrea Banino, Misha Denil, Ross Goroshin, Laurent Sifre, Koray Kavukcuoglu, et al. Learning to navigate in complex environments. In *Proceedings of the 5th International Conference on Learning Representations 2017*, 2016.

[134] Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. A Simple Neural Attentive Meta-Learner. *Int. Conf. Learn. Represent.*, 2018.

[135] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, 2016.

[136] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[137] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[138] Anastasios I Mourikis and Stergios I Roumeliotis. A multi-state constraint kalman filter for vision-aided inertial navigation. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 3565–3572. IEEE, 2007.

[139] R. Mur-Artal and J. D. Tardós. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, 2017.

[140] Raul Mur-Artal, Jose Maria Martinez Montiel, and Juan D Tardos. Orb-slam: a versatile and accurate monocular slam system. *IEEE Transactions on Robotics*, 31(5):1147–1163, 2015.

[141] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[142] Michael Neunert, Abbas Abdolmaleki, Markus Wulfmeier, Thomas Lampe, Nicolas Heess, Roland Hafner, and Martin Riedmiller. Continuous-Discrete Deep Reinforcement Learning for Hybrid Control in Robotics. *To appear in Conf. on Robot Learn. (CoRL)*, 2019.

[143] Richard A Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pages 127–136. IEEE, 2011.

[144] Alex Nichol, Joshua Achiam, and John Schulman. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*, 2018.

[145] David Nistér, Oleg Naroditsky, and James Bergen. Visual odometry for ground vehicle applications. *Journal of Field Robotics*, 23(1):3–20, 2006.

[146] J. Oh, V. Chockalingam, S. Singh, and H. Lee. Control of memory, active perception, and action in minecraft. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, 2016.

[147] Junhyuk Oh, Satinder Singh, and Honglak Lee. Value prediction network. In *Advances in Neural Information Processing Systems*, pages 6120–6130, 2017.

[148] E. Parisotto, J. Ba, and R. Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. In *Proceedings of the International Conference on Learning Representations, ICLR*, 2018.

[149] Emilio Parisotto, Jimmy Lei Ba, and Ruslan Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342*, 2015.

[150] Emilio Parisotto, Soham Ghosh, Sai Bhargav Yalamanchi, Varsha Chinnaobireddy, Yuhuai Wu, and Ruslan Salakhutdinov. Concurrent meta reinforcement learning. *arXiv preprint arXiv:1903.02710*, 2019.

[151] Emilio Parisotto and Ruslan Salakhutdinov. Neural map: Structured memory for deep reinforcement learning. *arXiv preprint arXiv:1702.08360*, 2017.

[152] Emilio Parisotto and Ruslan Salakhutdinov. Neural map: Structured memory for deep reinforcement learning. In *International Conference on Learning Representations*, 2018.

[153] Emilio Parisotto and Ruslan Salakhutdinov. Efficient transformers in reinforcement learning using actor-learner distillation. In *International Conference on Learning Representations*, 2021.

[154] Emilio Parisotto, Devendra Singh Chaplot, Jian Zhang, and Ruslan Salakhutdinov. Global pose estimation with an attention-based recurrent network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 237–246, 2018.

[155] Emilio Parisotto, H. Francis Song, Jack W. Rae, Razvan Pascanu, Caglar Gulcehre, Siddhant M. Jayakumar, Max Jaderberg, Raphael Lopez Kaufman, Aidan Clark, Seb Noury, Matthew M. Botvinick, Nicolas Heess, and Raia Hadsell. Stabilizing transformers for reinforcement learning. *arXiv preprent arXiv:1910.06764*, 2019.

[156] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning (ICML 2013)*, pages 1310–1318, 2013.

[157] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[158] Joelle Pineau, Geoff Gordon, and Sebastian Thrun. Point-based value iteration: an anytime algorithm for pomdps. In *Proceedings of the 18th international joint conference on Artificial intelligence*, pages 1025–1030. Morgan Kaufmann Publishers Inc., 2003.

[159] Lerrel Pinto, Marcin Andrychowicz, Peter Welinder, Wojciech Zaremba, and Pieter Abbeel. Asymmetric actor critic for image-based robot learning. *arXiv preprint arXiv:1710.06542*, 2017.

[160] Robert Clay Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

[161] Alexander Pritzel, Benigno Uria, Sriram Srinivasan, Adrià Puigdomènech Badia, Oriol Vinyals, Demis Hassabis, Daan Wierstra, and Charles Blundell. Neural episodic control. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2827–2836, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[162] Alexander Pritzel, Benigno Uria, Sriram Srinivasan, Adrià Puigdomènech, Oriol Vinyals, Demis Hassabis, Daan Wierstra, and Charles Blundell. Neural episodic control. *arXiv preprint arXiv:1703.01988*, 2017.

[163] Martin L Puterman. Markov decision processes: Discrete stochastic dynamic programming. 1994.

[164] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[165] J. W. Rae, J. J. Hunt, T. Harley, I. Danihelka, A. Senior, G. Wayne, A. Graves, and T. Lillicrap. Scaling memory-augmented neural networks with sparse reads and writes. *CoRR*, abs/1610.09027, 2016.

[166] Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, Chloe Hillier, and Timothy P. Lillicrap. Compressive transformers for long-range sequence modelling. In *International Conference on Learning Representations*, 2020.

[167] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[168] Larry A Rendell and David K Tcheng. Layered concept-learning and dynamically variable bias management.

[169] Samuel Ritter et al. *Meta-Reinforcement Learning with Episodic Recall: An Integrative Theory of Reward-Driven Learning.* PhD thesis, Princeton University, 2019.

[170] Samuel Ritter, Jane X Wang, Zeb Kurth-Nelson, Siddhant M Jayakumar, Charles Blundell, Razvan Pascanu, and Matthew Botvinick. Been there, done that: Meta-learning with episodic recall. *arXiv preprint arXiv:1805.09692*, 2018.

[171] Stéphane Ross, Geoffrey J Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *AISTATS*, volume 1, page 6, 2011.

[172] Stergios I. Roumeliotis and George A. Bekey. Bayesian estimation and kalman filtering: A unified framework for mobile robot localization, 2000.

[173] D. E. Rumelhart, G. E. Hinton, and J. L. McClelland. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter 2: A General Framework for Parallel Distributed Processing, pages 45–76. MIT Press, Cambridge, MA, USA, 1986.

[174] A. A. Rusu, S. G. Colmenarejo, C. Gulcehre, G. Desjardins, J. Kirkpatrick, R. Pascanu, V. Mnih, K. Kavukcuoglu, and R. Hadsell. Policy distillation. In *Proceedings of the International Conference on Learning Representations, ICLR"*, 2015.

[175] Andrei A. Rusu, Dushyant Rao, Jakub Sygnowski, Oriol Vinyals, Razvan Pascanu, Simon Osindero, and Raia Hadsell. Meta-learning with latent embedding optimization. In *International Conference on Learning Representations*, 2019.

[176] Renato F Salas-Moreno, Richard A Newcombe, Hauke Strasdat, Paul HJ Kelly, and Andrew J Davison. Slam++: Simultaneous localisation and mapping at the level of objects. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1352–1359, 2013.

[177] Adam Santoro, Ryan Faulkner, David Raposo, Jack Rae, Mike Chrzanowski, Theophane Weber, Daan Wierstra, Oriol Vinyals, Razvan Pascanu, and Timothy Lillicrap. Relational recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 7299–7310, 2018.

[178] Torsten Sattler, Bastian Leibe, and Leif Kobbelt. Efficient & effective prioritized matching for large-scale image-based localization. *IEEE transactions on pattern analysis and machine intelligence*, 39(9):1744–1756, 2016.

[179] Torsten Sattler, Will Maddern, Carl Toft, Akihiko Torii, Lars Hammarstrand, Erik Stenborg, Daniel Safari, Masatoshi Okutomi, Marc Pollefeys, Josef Sivic, et al. Benchmarking 6dof outdoor visual localization in changing conditions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8601–8610, 2018.

[180] T. Schaul and J. Schmidhuber. Metalearning. *Scholarpedia*, 5(6):4650, 2010. revision #91489.

[181] J. Schmidhuber. *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook.,*. PhD thesis, Institut für Informatik, Technische Universität München., 1987.

[182] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[183] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017.

[184] Hava T Siegelmann and Eduardo D Sontag. On the computational power of neural nets. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 440–449, 1992.

[185] David Silver, Leonard Newnham, David Barker, Suzanne Weller, and Jason McFall. Concurrent reinforcement learning from customer interactions. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 924–932, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[186] David Silver, Hado van Hasselt, Matteo Hessel, Tom Schaul, Arthur Guez, Tim Harley, Gabriel Dulac-Arnold, David Reichert, Neil Rabinowitz, Andre Barreto, et al. The predictron: End-to-end learning and planning. *arXiv preprint arXiv:1612.08810*, 2016.

[187] Randall Smith, Matthew Self, and Peter Cheeseman. Estimating uncertain spatial relationships in robotics. In *Autonomous robot vehicles*, pages 167–193. Springer, 1990.

[188] H. Francis Song, Abbas Abdolmaleki, Jost Tobias Springenberg, Aidan Clark, Hubert Soyer, Jack W. Rae, Seb Noury, Arun Ahuja, Siqi Liu, Dhruva Tirumala, Nicolas Heess, Dan Belov, Martin Riedmiller, and Matthew M. Botvinick. V-mpo: On-policy maximum a posteriori policy optimization for discrete and continuous control. In *International Conference on Learning Representations*, 2020.

[189] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

[190] Bradly C Stadie, Ge Yang, Rein Houthooft, Xi Chen, Yan Duan, Yuhuai Wu, Pieter Abbeel, and Ilya Sutskever. Some considerations on learning to explore via meta-reinforcement learning. *arXiv preprint arXiv:1803.01118*, 2018.

[191] Adam Stooke, Valentin Dalibard, Siddhant M Jayakumar, Wojciech M Czarnecki, and Max Jaderberg. Perception-prediction-reaction agents for deep reinforcement learning. *SPIRL Workshop*, 2019.

[192] Malcolm JA Strens. A bayesian framework for reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 943–950. Morgan Kaufmann Publishers Inc., 2000.

[193] Sainbayar Sukhbaatar, Rob Fergus, et al. Learning multiagent communication with backpropagation. In *Advances in Neural Information Processing Systems*, pages 2244–2252, 2016.

[194] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1139–1147, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[195] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[196] R. Sutton and A. Barto. *Reinforcement Learning: an Introduction.* MIT Press, 1998.

[197] Lei Tai, Giuseppe Paolo, and Ming Liu. Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 31–36. IEEE, 2017.

[198] Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. Value iteration networks. In *Advances in Neural Information Processing Systems*, pages 2146–2154, 2016.

[199] Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. Value Iteration Networks. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI 2017)*, pages 4949–4953, 2017.

[200] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *arXiv preprint arXiv:2009.06732*, 2020.

[201] Y. Teh, V. Bapst, W. M. Czarnecki, J. Quan, J. Kirkpatrick, R. Hadsell, N. Heess, and R. Pascanu. Distral: Robust multitask reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 4496–4506, 2017.

[202] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*, volume 1. MIT press Cambridge, 2005.

[203] Sebastian Thrun, Dieter Fox, Wolfram Burgard, and Frank Dellaert. Robust monte carlo localization for mobile robots. *Artificial intelligence*, 128(1-2):99–141, 2001.

[204] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. *arXiv preprint arXiv:1703.06907*, 2017.

[205] P. Utgoff. Shift of bias for inductive concept learning. *Machine Learning*, pages 163–190, 1986.

[206] Aaron Van den Oord, Nal Kalchbrenner, Lasse Espeholt, Oriol Vinyals, Alex Graves, and Koray Kavukcuoglu. Conditional image generation with pixelcnn decoders. In *Advances in neural information processing systems*, pages 4790–4798, 2016.

[207] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[208] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojtek Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, Timo Ewalds, Dan Horgan, Manuel Kroiss, Ivo Danihelka, John Agapiou, Junhyuk Oh, Valentin Dalibard, David Choi, Laurent Sifre, Yury Sulsky, Sasha Vezhnevets, James Molloy, Trevor Cai, David Budden, Tom Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Toby Pohlen, Dani Yogatama, Julia Cohen, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Chris Apps, Koray Kavukcuoglu, Demis Hassabis, and David Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II, 2019.

[209] Florian Walch, Caner Hazirbas, Laura Leal-Taixe, Torsten Sattler, Sebastian Hilsenbeck, and Daniel Cremers. Image-based localization using lstms for structured feature correlation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 627–637, 2017.

[210] Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016.

[211] Sen Wang, Ronald Clark, Hongkai Wen, and Niki Trigoni. Deepvo: Towards end-to-end visual odometry with deep recurrent convolutional neural networks. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pages 2043–2050. IEEE, 2017.

[212] Greg Wayne, Chia-Chun Hung, David Amos, Mehdi Mirza, Arun Ahuja, Agnieszka Grabska-Barwinska, Jack Rae, Piotr Mirowski, Joel Z. Leibo, Adam Santoro, Mevlana Gemici, Malcolm Reynolds, Tim Harley, Josh Abramson, Shakir Mohamed, Danilo Rezende, David Saxton, Adam Cain, Chloe Hillier, David Silver, Koray Kavukcuoglu, Matt Botvinick, Demis Hassabis, and Timothy Lillicrap. Unsupervised predictive memory in a goal-directed agent. *arXiv preprint arXiv:1803.10760*, 2018.

[213] Tobias Weyand, Ilya Kostrikov, and James Philbin. Planet-photo geolocation with convolutional neural networks. In *European Conference on Computer Vision*, pages 37–55. Springer, 2016.

[214] Thomas Whelan, Renato F Salas-Moreno, Ben Glocker, Andrew J Davison, and Stefan Leutenegger. Elasticfusion: Real-time dense slam and light source estimation. *The International Journal of Robotics Research*, 35(14):1697–1716, 2016.

[215] Yuhuai Wu, Saizheng Zhang, Ying Zhang, Yoshua Bengio, and Ruslan R Salakhutdinov. On multiplicative integration with recurrent neural networks. In *Advances in neural information processing systems*, pages 2856–2864, 2016.

[216] Marek Wydmuch, Michał Kempka, and Wojciech Jaśkowski. Vizdoom competitions: Playing doom from pixels. *IEEE Transactions on Games*, 2018.

[217] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tieyan Liu. On layer normalization in the transformer architecture. In *International Conference on Machine Learning*, pages 10524–10533. PMLR, 2020.

[218] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in Neural Information Processing Systems*, 2019.

[219] Vinicius Zambaldi, David Raposo, Adam Santoro, Victor Bapst, Yujia Li, Igor Babuschkin, Karl Tuyls, David Reichert, Timothy Lillicrap, Edward Lockhart, Murray Shanahan, Victoria Langston, Razvan Pascanu, Matthew Botvinick, Oriol Vinyals, and Peter Battaglia. Deep reinforcement learning with relational inductive biases. In *International Conference on Learning Representations*, 2019.

[220] Jingwei Zhang, Jost Tobias Springenberg, Joschka Boedecker, and Wolfram Burgard. Deep reinforcement learning with successor features for navigation across similar environments. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2371–2378. IEEE, 2017.

[221] Jingwei Zhang, Lei Tai, Joschka Boedecker, Wolfram Burgard, and Ming Liu. Neural slam: Learning to explore with external memory. *arXiv preprint arXiv:1706.09520*, 2017.

[222] Nevin Lianwen Zhang and Weihong Zhang. Speeding up the convergence of value iteration in partially observable markov decision processes. *Journal of Artificial Intelligence Research*, 14:29–51, 2001.

[223] Yuke Zhu, Roozbeh Mottaghi, Eric Kolve, Joseph J Lim, Abhinav Gupta, Li Fei-Fei, and Ali Farhadi. Target-driven visual navigation in indoor scenes using deep reinforcement learning. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 3357–3364. IEEE, 2017.

[224] Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutník, and Jürgen Schmidhuber. Recurrent highway networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 4189–4198. JMLR. org, 2017.

[225] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *ICLR*, 2017.